

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

```

```

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

```

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

```

1  using System;
2  using System.Collections.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Decorators
7  {
8      public class LinksItselfConstantToSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ↳ EqualityComparer<TLink>.Default;
12
13         public LinksItselfConstantToSelfReferenceResolver(ILinks<TLink> links) : base(links) { }
14
15         public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
16         {
17             var constants = Constants;
18             var itselfConstant = constants.Itself;
19             var indexPartConstant = constants.IndexPart;
20             var sourcePartConstant = constants.SourcePart;
21             var targetPartConstant = constants.TargetPart;
22             var restrictionsCount = restrictions.Count;
23             if (!_equalityComparer.Equals(constants.Any, itselfConstant)
24                 && ((restrictionsCount > indexPartConstant) &&
25                     ↳ _equalityComparer.Equals(restrictions[indexPartConstant], itselfConstant))
26                 || ((restrictionsCount > sourcePartConstant) &&
27                     ↳ _equalityComparer.Equals(restrictions[sourcePartConstant], itselfConstant))
28                 || ((restrictionsCount > targetPartConstant) &&
29                     ↳ _equalityComparer.Equals(restrictions[targetPartConstant], itselfConstant))))
30             {
31                 // Itself constant is not supported for Each method right now, skipping execution
32                 return constants.Continue;
33             }
34             return Links.Each(handler, restrictions);
35         }
36
37         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
38             ↳ Links.Update(restrictions, Links.ResolveConstantAsSelfReference(Constants.Itself,
39                 ↳ restrictions, substitution));
40     }
41 }

```

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

```

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

```

```

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

```

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

```

1 using System.Collections.Generic;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Decorators
6 {
7     public class LinksNullConstantToSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
8     {
9         public LinksNullConstantToSelfReferenceResolver(ILinks<TLink> links) : base(links) { }
10
11         public override TLink Create(IList<TLink> restrictions)
12         {
13             var link = Links.Create();
14             return Links.Update(link, link, link);
15         }
16
17         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
18             ↳ Links.Update(restrictions, Links.ResolveConstantAsSelfReference(Constants.Null,
19             ↳ restrictions, substitution));
20     }
21 }

```

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

```

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

```

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

```

1 using System.Collections.Generic;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Decorators
6 {
7     public class LinksUniquenessValidator<TLink> : LinksDecoratorBase<TLink>

```

```

8     {
9         public LinksUniquenessValidator(ILinks<TLink> links) : base(links) { }
10
11        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
12        {
13            Links.EnsureDoesNotExists(substitution[Constants.SourcePart],
14            ↪ substitution[Constants.TargetPart]);
15            return Links.Update(restrictions, substitution);
16        }
17    }

```

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

```

1  using System.Collections.Generic;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Decorators
6  {
7      public class LinksUsagesValidator<TLink> : LinksDecoratorBase<TLink>
8      {
9          public LinksUsagesValidator(ILinks<TLink> links) : base(links) { }
10
11         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
12         {
13             Links.EnsureNoUsages(restrictions[Constants.IndexPart]);
14             return Links.Update(restrictions, substitution);
15         }
16
17         public override void Delete(IList<TLink> restrictions)
18         {
19             var link = restrictions[Constants.IndexPart];
20             Links.EnsureNoUsages(link);
21             Links.Delete(link);
22         }
23     }
24 }

```

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

```

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

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Collections;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Decorators
8  {
9      /// <summary>
10     /// Представляет объект для работы с базой данных (файлом) в формате Links (массива связей).
11     /// </summary>
12     /// <remarks>
13     /// Возможные оптимизации:
14     /// Объединение в одном поле Source и Target с уменьшением до 32 бит.
15     ///     + меньше объём БД
16     ///     - меньше производительность
17     ///     - больше ограничение на количество связей в БД)
18     /// Ленивое хранение размеров поддеревьев (расчитываемое по мере использования БД)
19     ///     + меньше объём БД
20     ///     - больше сложность

```

```

21 ///
22 /// Текущее теоретическое ограничение на индекс связи, из-за использования 5 бит в размере
    ↳ поддеревьев для AVL баланса и флагов нитей: 2 в степени(64 минус 5 равно 59 ) равно 576
    ↳ 460 752 303 423 488
23 /// Желательно реализовать поддержку переключения между деревьями и битовыми индексами
    ↳ (битовыми строками) - вариант матрицы (выстраиваемой лениво).
24 ///
25 /// Решить отключать ли проверки при компиляции под Release. Т.е. исключения будут
    ↳ выбрасываться только при #if DEBUG
26 /// </remarks>
27 public class UInt64Links : LinksDisposableDecoratorBase<ulong>
28 {
29     public UInt64Links(ILinks<ulong> links) : base(links) { }
30
31     public override ulong Each(Func<IList<ulong>, ulong> handler, IList<ulong> restrictions)
32     {
33         this.EnsureLinkIsAnyOrExists(restrictions);
34         return Links.Each(handler, restrictions);
35     }
36
37     public override ulong Create(IList<ulong> restrictions) => Links.CreatePoint();
38
39     public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
40     {
41         var constants = Constants;
42         var nullConstant = constants.Null;
43         if (restrictions.IsNullOrEmpty())
44         {
45             return nullConstant;
46         }
47         // TODO: Looks like this is a common type of exceptions linked with restrictions
48         ↳ support
49         if (substitution.Count != 3)
50         {
51             throw new NotSupportedException();
52         }
53         var indexPartConstant = constants.IndexPart;
54         var updatedLink = restrictions[indexPartConstant];
55         this.EnsureLinkExists(updatedLink,
56             ↳ $"{nameof(restrictions)}[{nameof(indexPartConstant)}]");
57         var sourcePartConstant = constants.SourcePart;
58         var newSource = substitution[sourcePartConstant];
59         this.EnsureLinkIsItselfOrExists(newSource,
60             ↳ $"{nameof(substitution)}[{nameof(sourcePartConstant)}]");
61         var targetPartConstant = constants.TargetPart;
62         var newTarget = substitution[targetPartConstant];
63         this.EnsureLinkIsItselfOrExists(newTarget,
64             ↳ $"{nameof(substitution)}[{nameof(targetPartConstant)}]");
65         var existedLink = nullConstant;
66         var itselfConstant = constants.Itself;
67         if (newSource != itselfConstant && newTarget != itselfConstant)
68         {
69             existedLink = this.SearchOrDefault(newSource, newTarget);
70         }
71         if (existedLink == nullConstant)
72         {
73             var before = Links.GetLink(updatedLink);
74             if (before[sourcePartConstant] != newSource || before[targetPartConstant] !=
75                 ↳ newTarget)
76             {
77                 Links.Update(updatedLink, newSource == itselfConstant ? updatedLink :
78                     ↳ newSource,
79                             newTarget == itselfConstant ? updatedLink :
80                                 ↳ newTarget);
81             }
82             return updatedLink;
83         }
84         else
85         {
86             return this.MergeAndDelete(updatedLink, existedLink);
87         }
88     }
89
90     public override void Delete(IList<ulong> restrictions)
91     {
92         var linkIndex = restrictions[Constants.IndexPart];
93         Links.EnsureLinkExists(linkIndex);
94         Links.EnforceResetValues(linkIndex);
95     }
96 }

```

```

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

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using Platform.Collections;
5  using Platform.Collections.Arrays;
6  using Platform.Collections.Lists;
7  using Platform.Data.Universal;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets.Decorators
12 {
13     /// <remarks>
14     /// What does empty pattern (for condition or substitution) mean? Nothing or Everything?
15     /// Now we go with nothing. And nothing is something one, but empty, and cannot be changed
16     /// ↪ by itself. But can cause creation (update from nothing) or deletion (update to nothing).
17     ///
18     /// TODO: Decide to change to IDoubletLinks or not to change. (Better to create
19     /// ↪ DefaultUniLinksBase, that contains logic itself and can be implemented using both
20     /// ↪ IDoubletLinks and ILinks.)
21     /// </remarks>
22     internal class UniLinks<TLink> : LinksDecoratorBase<TLink>, IUniLinks<TLink>
23     {
24         private static readonly EqualityComparer<TLink> _equalityComparer =
25             ↪ EqualityComparer<TLink>.Default;
26
27         public UniLinks(ILinks<TLink> links) : base(links) { }
28
29         private struct Transition
30         {
31             public IList<TLink> Before;
32             public IList<TLink> After;
33
34             public Transition(IList<TLink> before, IList<TLink> after)
35             {
36                 Before = before;
37                 After = after;
38             }
39         }
40
41         //public static readonly TLink NullConstant = Use<LinksConstants<TLink>>.Single.Null;
42         //public static readonly IReadOnlyList<TLink> NullLink = new
43         ↪ ReadOnlyCollection<TLink>(new List<TLink> { NullConstant, NullConstant, NullConstant
44         ↪ });
45
46         // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution
47         ↪ (Links-Expression)
48         public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>, TLink>
49         ↪ matchedHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
50         ↪ substitutedHandler)
51         {
52             ///List<Transition> transitions = null;
53             ///if (!restriction.IsNullOrEmpty())
54             ///{
55             ///    // Есть причина делать проход (чтение)
56             ///    if (matchedHandler != null)
57             ///    {
58             ///        if (!substitution.IsNullOrEmpty())
59             ///        {
60             ///            // restriction => { 0, 0, 0 } | { 0 } // Create
61             ///            // substitution => { itself, 0, 0 } | { itself, itself, itself } //
62             ↪ Create / Update
63             ///            // substitution => { 0, 0, 0 } | { 0 } // Delete
64             ///            transitions = new List<Transition>();
65             ///            if (Equals(substitution[Constants.IndexPart], Constants.Null))
66             ///            {
67             ///                // If index is Null, that means we always ignore every other
68             ↪ value (they are also Null by definition)
69             ///            var matchDecision = matchedHandler(, NullLink);
70             ///            if (Equals(matchDecision, Constants.Break))
71             ///                return false;
72             ///            if (!Equals(matchDecision, Constants.Skip))

```

```

62         transitions.Add(new Transition(matchedLink, newValue));
63     }
64     else
65     {
66         Func<T, bool> handler;
67         handler = link =>
68         {
69             var matchedLink = Memory.GetLinkValue(link);
70             var newValue = Memory.GetLinkValue(link);
71             newValue[Constants.IndexPart] = Constants.Itself;
72             newValue[Constants.SourcePart] =
↳ Equals(substitution[Constants.SourcePart], Constants.Itself) ?
↳ matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
73             newValue[Constants.TargetPart] =
↳ Equals(substitution[Constants.TargetPart], Constants.Itself) ?
↳ matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
74             var matchDecision = matchedHandler(matchedLink, newValue);
75             if (Equals(matchDecision, Constants.Break))
76                 return false;
77             if (!Equals(matchDecision, Constants.Skip))
78                 transitions.Add(new Transition(matchedLink, newValue));
79             return true;
80         };
81         if (!Memory.Each(handler, restriction))
82             return Constants.Break;
83     }
84 }
85 else
86 {
87     Func<T, bool> handler = link =>
88     {
89         var matchedLink = Memory.GetLinkValue(link);
90         var matchDecision = matchedHandler(matchedLink, matchedLink);
91         return !Equals(matchDecision, Constants.Break);
92     };
93     if (!Memory.Each(handler, restriction))
94         return Constants.Break;
95 }
96 }
97 else
98 {
99     if (substitution != null)
100     {
101         transitions = new List<IList<T>>();
102         Func<T, bool> handler = link =>
103         {
104             var matchedLink = Memory.GetLinkValue(link);
105             transitions.Add(matchedLink);
106             return true;
107         };
108         if (!Memory.Each(handler, restriction))
109             return Constants.Break;
110     }
111     else
112     {
113         return Constants.Continue;
114     }
115 }
116 }
117 if (substitution != null)
118 {
119     // Есть причина делать замену (запись)
120     if (substitutedHandler != null)
121     {
122     }
123     else
124     {
125     }
126 }
127 return Constants.Continue;
128
129 //if (restriction.IsNullOrEmpty()) // Create
130 //{
131 //    substitution[Constants.IndexPart] = Memory.AllocateLink();
132 //    Memory.SetLinkValue(substitution);
133 //}
134 //else if (substitution.IsNullOrEmpty()) // Delete

```



```

135     //{
136     //     Memory.FreeLink(restriction[Constants.IndexPart]);
137     //}
138     //else if (restriction.EqualTo(substitution)) // Read or ("repeat" the state) // Each
139     //{
140     //     // No need to collect links to list
141     //     // Skip == Continue
142     //     // No need to check substitutedHandler
143     //     if (!Memory.Each(link => !Equals(matchedHandler(Memory.GetLinkValue(link)),
144     ↪ Constants.Break), restriction))
145     //         return Constants.Break;
146     //}
147     //else // Update
148     //{
149     //     //List<IList<T>> matchedLinks = null;
150     //     if (matchedHandler != null)
151     //     {
152     //         matchedLinks = new List<IList<T>>();
153     //         Func<T, bool> handler = link =>
154     //         {
155     //             var matchedLink = Memory.GetLinkValue(link);
156     //             var matchDecision = matchedHandler(matchedLink);
157     //             if (Equals(matchDecision, Constants.Break))
158     //                 return false;
159     //             if (!Equals(matchDecision, Constants.Skip))
160     //                 matchedLinks.Add(matchedLink);
161     //             return true;
162     //         };
163     //         if (!Memory.Each(handler, restriction))
164     //             return Constants.Break;
165     //     }
166     //     if (!matchedLinks.IsNullOrEmpty())
167     //     {
168     //         var totalMatchedLinks = matchedLinks.Count;
169     //         for (var i = 0; i < totalMatchedLinks; i++)
170     //         {
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(IList<TLink> patternOrCondition, Func<IList<TLink>, TLink>
199 ↪ matchHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
200 ↪ substitutionHandler)
201 {
202     if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
203     {
204         return Constants.Continue;
205     }
206     else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO:
207     ↪ Check if it is a correct condition
208     {
209         // Or it only applies to trigger without matchHandler.
210         throw new NotImplementedException();
211     }
212     else if (!substitution.IsNullOrEmpty()) // Creation

```

```

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

```


./Platform.Data.Doubles/Douplet.cs

```
1 using System;
2 using System.Collections.Generic;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doubles
7 {
8     public struct Douplet<T> : IEquatable<Douplet<T>>
9     {
10         private static readonly EqualityComparer<T> _equalityComparer =
11             ↳ EqualityComparer<T>.Default;
12
13         public T Source { get; set; }
14         public T Target { get; set; }
15
16         public Douplet(T source, T target)
17         {
18             Source = source;
19             Target = target;
20         }
21
22         public override string ToString() => $"{Source}->{Target}";
23
24         public bool Equals(Douplet<T> other) => _equalityComparer.Equals(Source, other.Source)
25             ↳ && _equalityComparer.Equals(Target, other.Target);
26
27         public override bool Equals(object obj) => obj is Douplet<T> doublet ?
28             ↳ base.Equals(doublet) : false;
29
30         public override int GetHashCode() => (Source, Target).GetHashCode();
31     }
32 }
```

./Platform.Data.Doubles/Hybrid.cs

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

```

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

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

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

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

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

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

```

```

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

```

./Platform.Data.Doublets/ILinks.cs

```

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

```

./Platform.Data.Doublets/ILinksExtensions.cs

```

1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System.Linq;
5  using System.Runtime.CompilerServices;
6  using Platform.Ranges;
7  using Platform.Collections.Arrays;
8  using Platform.Numbers;
9  using Platform.Random;
10 using Platform.Setters;
11 using Platform.Data.Exceptions;
12 using Platform.Data.Doublets.Decorators;
13
14 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16 namespace Platform.Data.Doublets
17 {
18     public static class ILinksExtensions
19     {
20         public static void RunRandomCreations<TLink>(this ILinks<TLink> links, long
21             ↪ amountOfCreations)
22         {
23             for (long i = 0; i < amountOfCreations; i++)
24             {
25                 var linksAddressRange = new Range<ulong>(0, (Integer<TLink>)links.Count());
26                 Integer<TLink> source = RandomHelpers.Default.NextUInt64(linksAddressRange);
27                 Integer<TLink> target = RandomHelpers.Default.NextUInt64(linksAddressRange);
28                 links.CreateAndUpdate(source, target);
29             }
30
31             public static void RunRandomSearches<TLink>(this ILinks<TLink> links, long
32                 ↪ amountOfSearches)
33             {
34                 for (long i = 0; i < amountOfSearches; i++)
35                 {
36                     var linkAddressRange = new Range<ulong>(1, (Integer<TLink>)links.Count());

```

```

36         Integer<TLink> source = RandomHelpers.Default.NextUInt64(linkAddressRange);
37         Integer<TLink> target = RandomHelpers.Default.NextUInt64(linkAddressRange);
38         links.SearchOrDefault(source, target);
39     }
40 }
41
42 public static void RunRandomDeletions<TLink>(this ILinks<TLink> links, long
    ↳ amountOfDeletions)
43 {
44     var min = (ulong)amountOfDeletions > (Integer<TLink>)links.Count() ? 1 :
    ↳ (Integer<TLink>)links.Count() - (ulong)amountOfDeletions;
45     for (long i = 0; i < amountOfDeletions; i++)
46     {
47         var linksAddressRange = new Range<ulong>(min, (Integer<TLink>)links.Count());
48         Integer<TLink> link = RandomHelpers.Default.NextUInt64(linksAddressRange);
49         links.Delete(link);
50         if ((Integer<TLink>)links.Count() < min)
51         {
52             break;
53         }
54     }
55 }
56
57 public static void Delete<TLink>(this ILinks<TLink> links, TLink linkToDelete) =>
    ↳ links.Delete(new LinkAddress<TLink>(linkToDelete));
58
59 /// <remarks>
60 /// TODO: Возможно есть очень простой способ это сделать.
61 /// (Например просто удалить файл, или изменить его размер таким образом,
62 /// чтобы удалится 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),
92                 ↪ nameof(index));
93             if (index == _constants.IndexPart)
94             {
95                 return Index;
96             }
97             if (index == _constants.SourcePart)
98             {
99                 return Source;
100             }
101             if (index == _constants.TargetPart)
102             {
103                 return Target;
104             }
105             throw new NotSupportedException(); // Impossible path due to
106             ↪ Ensure.ArgumentInRange
107         }
108         set => throw new NotSupportedException();
109     }

```

```

109     IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
110
111     public IEnumerator<TLink> GetEnumerator()
112     {
113         yield return Index;
114         yield return Source;
115         yield return Target;
116     }
117
118     public void Add(TLink item) => throw new NotSupportedException();
119
120     public void Clear() => throw new NotSupportedException();
121
122     public bool Contains(TLink item) => IndexOf(item) >= 0;
123
124     public void CopyTo(TLink[] array, int arrayIndex)
125     {
126         Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
127         Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
128             ↪ nameof(arrayIndex));
129         if (arrayIndex + Length > array.Length)
130         {
131             throw new InvalidOperationException();
132         }
133         array[arrayIndex++] = Index;
134         array[arrayIndex++] = Source;
135         array[arrayIndex] = Target;
136     }
137
138     public bool Remove(TLink item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
139
140     public int IndexOf(TLink item)
141     {
142         if (_equalityComparer.Equals(Index, item))
143         {
144             return _constants.IndexPart;
145         }
146         if (_equalityComparer.Equals(Source, item))
147         {
148             return _constants.SourcePart;
149         }
150         if (_equalityComparer.Equals(Target, item))
151         {
152             return _constants.TargetPart;
153         }
154         return -1;
155     }
156
157     public void Insert(int index, TLink item) => throw new NotSupportedException();
158
159     public void RemoveAt(int index) => throw new NotSupportedException();
160
161     #endregion
162 }

```

./Platform.Data.Doublets/LinkExtensions.cs

```

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

```

./Platform.Data.Doublets/LinksOperatorBase.cs

```

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

```

```

9     }
10 }

```

./Platform.Data.Doublets/Numbers/Raw/AddressToRawNumberConverter.cs

```

1 using Platform.Interfaces;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Numbers.Raw
6 {
7     public class AddressToRawNumberConverter<TLink> : IConverter<TLink>
8     {
9         public TLink Convert(TLink source) => new Hybrid<TLink>(source, isExternal: true);
10    }
11 }

```

./Platform.Data.Doublets/Numbers/Raw/RawNumberToAddressConverter.cs

```

1 using Platform.Interfaces;
2 using Platform.Numbers;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Numbers.Raw
7 {
8     public class RawNumberToAddressConverter<TLink> : IConverter<TLink>
9     {
10        public TLink Convert(TLink source) => (Integer<TLink>)new
            ↳ Hybrid<TLink>(source).AbsoluteValue;
11    }
12 }

```

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

```

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

```

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

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Data.Doublets.ResizableDirectMemory
4  {
5      public interface ILinksListMethods<TLink>
6      {
7          void Detach(TLink freeLink);
8          void AttachAsFirst(TLink link);
9      }
10 }

```

./Platform.Data.Doublets/ResizableDirectMemory/ILinksTreeMethods.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.ResizableDirectMemory
7  {
8      public interface ILinksTreeMethods<TLink>
9      {
10         TLink CountUsages(TLink link);
11         TLink Search(TLink source, TLink target);
12         TLink EachUsage(TLink source, Func<IList<TLink>, TLink> handler);
13         void Detach(ref TLink firstAsSource, TLink linkIndex);
14         void Attach(ref TLink firstAsSource, TLink linkIndex);
15     }
16 }

```

./Platform.Data.Doublets/ResizableDirectMemory/LinksAVLBalancedTreeMethodsBase.cs

```

1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Numbers;
6  using Platform.Collections.Methods.Trees;
7  using static System.Runtime.CompilerServices.Unsafe;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets.ResizableDirectMemory
12 {
13     public unsafe abstract class LinksAVLBalancedTreeMethodsBase<TLink> :
14         ↪ SizedAndThreadedAVLBalancedTreeMethods<TLink>
15     {
16         private readonly ResizableDirectMemoryLinks<TLink> _memory;
17         private readonly LinksConstants<TLink> _constants;
18         protected readonly byte* Links;
19         protected readonly byte* Header;
20
21         public LinksAVLBalancedTreeMethodsBase(ResizableDirectMemoryLinks<TLink> memory, byte*
22             ↪ links, byte* header)
23         {
24             Links = links;
25             Header = header;
26             _memory = memory;
27             _constants = memory.Constants;
28         }
29     }

```

```

28 [MethodImpl(MethodImplOptions.AggressiveInlining)]
29 protected abstract TLink GetTreeRoot();
30
31 [MethodImpl(MethodImplOptions.AggressiveInlining)]
32 protected abstract TLink GetBasePartValue(TLink link);
33
34 [MethodImpl(MethodImplOptions.AggressiveInlining)]
35 protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
    ↪ rootSource, TLink rootTarget);
36
37 [MethodImpl(MethodImplOptions.AggressiveInlining)]
38 protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
    ↪ rootSource, TLink rootTarget);
39
40 [MethodImpl(MethodImplOptions.AggressiveInlining)]
41 internal virtual ref LinksHeader<TLink> GetHeaderReference() => ref
    ↪ AsRef<LinksHeader<TLink>>(Header);
42
43 [MethodImpl(MethodImplOptions.AggressiveInlining)]
44 internal virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
    ↪ AsRef<RawLink<TLink>>((void*)(Links + RawLink<TLink>.SizeInBytes *
    ↪ (Integer<TLink>)link));
45
46 [MethodImpl(MethodImplOptions.AggressiveInlining)]
47 protected virtual Link<TLink> GetLinkValues(TLink current) =>
    ↪ _memory.GetLinkStruct(current);
48
49 [MethodImpl(MethodImplOptions.AggressiveInlining)]
50 protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
51 {
52     ref var firstLink = ref GetLinkReference(first);
53     ref var secondLink = ref GetLinkReference(second);
54     return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
    ↪ secondLink.Source, secondLink.Target);
55 }
56
57 [MethodImpl(MethodImplOptions.AggressiveInlining)]
58 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
59 {
60     ref var firstLink = ref GetLinkReference(first);
61     ref var secondLink = ref GetLinkReference(second);
62     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
    ↪ secondLink.Source, secondLink.Target);
63 }
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 protected static TLink GetSizeValue(TLink value) => Bit<TLink>.PartialRead(value, 5, -5);
67
68 [MethodImpl(MethodImplOptions.AggressiveInlining)]
69 protected static void SetSizeValue(ref TLink storedValue, TLink size) => storedValue =
    ↪ Bit<TLink>.PartialWrite(storedValue, size, 5, -5);
70
71 [MethodImpl(MethodImplOptions.AggressiveInlining)]
72 protected bool GetLeftIsChildValue(TLink value)
73 {
74     unchecked
75     {
76         //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 4, 1);
77         return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 4, 1), default);
78     }
79 }
80
81 [MethodImpl(MethodImplOptions.AggressiveInlining)]
82 protected static void SetLeftIsChildValue(ref TLink storedValue, bool value)
83 {
84     unchecked
85     {
86         var previousValue = storedValue;
87         var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 4,
    ↪ 1);
88         storedValue = modified;
89     }
90 }
91
92 [MethodImpl(MethodImplOptions.AggressiveInlining)]
93 protected bool GetRightIsChildValue(TLink value)
94 {
95     unchecked

```

```

96     {
97         //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 3, 1);
98         return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 3, 1), default);
99     }
100 }
101
102 [MethodImpl(MethodImplOptions.AggressiveInlining)]
103 protected static void SetRightIsChildValue(ref TLink storedValue, bool value)
104 {
105     unchecked
106     {
107         var previousValue = storedValue;
108         var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 3,
109             ↪ 1);
110         storedValue = modified;
111     }
112 }
113
114 [MethodImpl(MethodImplOptions.AggressiveInlining)]
115 protected static sbyte GetBalanceValue(TLink storedValue)
116 {
117     unchecked
118     {
119         var value = (int)(Integer<TLink>)Bit<TLink>.PartialRead(storedValue, 0, 3);
120         value |= 0xF8 * ((value & 4) >> 2); // if negative, then continue ones to the
121             ↪ end of sbyte
122         return (sbyte)value;
123     }
124 }
125
126 [MethodImpl(MethodImplOptions.AggressiveInlining)]
127 protected static void SetBalanceValue(ref TLink storedValue, sbyte value)
128 {
129     unchecked
130     {
131         var packagedValue = (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) | value &
132             ↪ 3);
133         var modified = Bit<TLink>.PartialWrite(storedValue, packagedValue, 0, 3);
134         storedValue = modified;
135     }
136 }
137
138 public TLink this[TLink index]
139 {
140     get
141     {
142         var root = GetTreeRoot();
143         if (GreaterOrEqualThan(index, GetSize(root)))
144         {
145             return Zero;
146         }
147         while (!EqualToZero(root))
148         {
149             var left = GetLeftOrDefault(root);
150             var leftSize = GetSizeOrZero(left);
151             if (LessThan(index, leftSize))
152             {
153                 root = left;
154                 continue;
155             }
156             if (IsEquals(index, leftSize))
157             {
158                 return root;
159             }
160             root = GetRightOrDefault(root);
161             index = Subtract(index, Increment(leftSize));
162         }
163         return Zero; // TODO: Impossible situation exception (only if tree structure
164             ↪ broken)
165     }
166 }
167
168 /// <summary>
169 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
170 /// ↪ (концом).
171 /// </summary>
172 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
173 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
174 /// <returns>Индекс искомой связи.</returns>

```

```

170 public TLink Search(TLink source, TLink target)
171 {
172     var root = GetTreeRoot();
173     while (!EqualToZero(root))
174     {
175         ref var rootLink = ref GetLinkReference(root);
176         var rootSource = rootLink.Source;
177         var rootTarget = rootLink.Target;
178         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
179             ↪ node.Key < root.Key
180         {
181             root = GetLeftOrDefault(root);
182         }
183         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
184             ↪ node.Key > root.Key
185         {
186             root = GetRightOrDefault(root);
187         }
188         else // node.Key == root.Key
189         {
190             return root;
191         }
192     }
193     return Zero;
194 }
195 // TODO: Return indices range instead of references count
196 public TLink CountUsages(TLink link)
197 {
198     var root = GetTreeRoot();
199     var total = GetSize(root);
200     var totalRightIgnore = Zero;
201     while (!EqualToZero(root))
202     {
203         var @base = GetBasePartValue(root);
204         if (LessOrEqualThan(@base, link))
205         {
206             root = GetRightOrDefault(root);
207         }
208         else
209         {
210             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
211             root = GetLeftOrDefault(root);
212         }
213     }
214     root = GetTreeRoot();
215     var totalLeftIgnore = Zero;
216     while (!EqualToZero(root))
217     {
218         var @base = GetBasePartValue(root);
219         if (GreaterOrEqualThan(@base, link))
220         {
221             root = GetLeftOrDefault(root);
222         }
223         else
224         {
225             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
226             root = GetRightOrDefault(root);
227         }
228     }
229     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
230 }
231
232 public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
233 {
234     var root = GetTreeRoot();
235     if (EqualToZero(root))
236     {
237         return _constants.Continue;
238     }
239     TLink first = Zero, current = root;
240     while (!EqualToZero(current))
241     {
242         var @base = GetBasePartValue(current);
243         if (GreaterOrEqualThan(@base, link))
244         {
245             if (IsEquals(@base, link))
246             {

```

```

247         first = current;
248     }
249     current = GetLeftOrDefault(current);
250 }
251 else
252 {
253     current = GetRightOrDefault(current);
254 }
255 }
256 if (!EqualToZero(first))
257 {
258     current = first;
259     while (true)
260     {
261         if (IsEquals(handler(GetLinkValues(current)), _constants.Break))
262         {
263             return _constants.Break;
264         }
265         current = GetNext(current);
266         if (EqualToZero(current) || !IsEquals(GetBasePartValue(current), link))
267         {
268             break;
269         }
270     }
271 }
272 return _constants.Continue;
273 }
274
275 protected override void PrintNodeValue(TLink node, StringBuilder sb)
276 {
277     ref var link = ref GetLinkReference(node);
278     sb.Append(' ');
279     sb.Append(link.Source);
280     sb.Append('-');
281     sb.Append('>');
282     sb.Append(link.Target);
283 }
284 }
285 }

```

./Platform.Data.Doublets/ResizableDirectMemory/LinksHeader.cs

```

1 using Platform.Unsafe;
2
3 namespace Platform.Data.Doublets.ResizableDirectMemory
4 {
5     internal struct LinksHeader<TLink>
6     {
7         public static readonly long SizeInBytes = Structure<LinksHeader<TLink>>.Size;
8
9         public TLink AllocatedLinks;
10        public TLink ReservedLinks;
11        public TLink FreeLinks;
12        public TLink FirstFreeLink;
13        public TLink FirstAsSource;
14        public TLink FirstAsTarget;
15        public TLink LastFreeLink;
16        public TLink Reserved8;
17    }
18 }

```

./Platform.Data.Doublets/ResizableDirectMemory/LinksSourcesAVLBalancedTreeMethods.cs

```

1 using System.Runtime.CompilerServices;
2 using Platform.Numbers;
3 using static System.Runtime.CompilerServices.Unsafe;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.ResizableDirectMemory
8 {
9     public unsafe class LinksSourcesAVLBalancedTreeMethods<TLink> :
10         ↳ LinksAVLBalancedTreeMethodsBase<TLink>, ILinksTreeMethods<TLink>
11     {
12         public LinksSourcesAVLBalancedTreeMethods(ResizableDirectMemoryLinks<TLink> memory,
13             ↳ byte* links, byte* header) : base(memory, links, header) { }
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         protected unsafe override ref TLink GetLeftReference(TLink node) => ref
17             ↳ GetLinkReference(node).LeftAsSource;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

17     protected unsafe override ref TLink GetRightReference(TLink node) => ref
18         ↳ GetLinkReference(node).RightAsSource;
19
20     [MethodImpl(MethodImplOptions.AggressiveInlining)]
21     protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsSource;
22
23     [MethodImpl(MethodImplOptions.AggressiveInlining)]
24     protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsSource;
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     protected override void SetLeft(TLink node, TLink left) =>
28         ↳ GetLinkReference(node).LeftAsSource = left;
29
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     protected override void SetRight(TLink node, TLink right) =>
32         ↳ GetLinkReference(node).RightAsSource = right;
33
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     protected override TLink GetSize(TLink node) =>
36         ↳ GetSizeValue(GetLinkReference(node).SizeAsSource);
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
40         ↳ GetLinkReference(node).SizeAsSource, size);
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     protected override bool GetLeftIsChild(TLink node) =>
44         ↳ GetLeftIsChildValue(GetLinkReference(node).SizeAsSource);
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected override void SetLeftIsChild(TLink node, bool value) =>
48         ↳ SetLeftIsChildValue(ref GetLinkReference(node).SizeAsSource, value);
49
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     protected override bool GetRightIsChild(TLink node) =>
52         ↳ GetRightIsChildValue(GetLinkReference(node).SizeAsSource);
53
54     [MethodImpl(MethodImplOptions.AggressiveInlining)]
55     protected override void SetRightIsChild(TLink node, bool value) =>
56         ↳ SetRightIsChildValue(ref GetLinkReference(node).SizeAsSource, value);
57
58     [MethodImpl(MethodImplOptions.AggressiveInlining)]
59     protected override sbyte GetBalance(TLink node) =>
60         ↳ GetBalanceValue(GetLinkReference(node).SizeAsSource);
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
64         ↳ GetLinkReference(node).SizeAsSource, value);
65
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsSource;
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;
71
72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
74         ↳ TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
75         ↳ (IsEquals(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
76
77     [MethodImpl(MethodImplOptions.AggressiveInlining)]
78     protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
79         ↳ TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
80         ↳ (IsEquals(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
81
82     [MethodImpl(MethodImplOptions.AggressiveInlining)]
83     protected override void ClearNode(TLink node)
84     {
85         ref var link = ref GetLinkReference(node);
86         link.LeftAsSource = Zero;
87         link.RightAsSource = Zero;
88         link.SizeAsSource = Zero;
89     }
89 }
90 }

```

./Platform.Data.Doublets/ResizableDirectMemory/LinksTargetsAVLBalancedTreeMethods.cs

```
1 using System.Runtime.CompilerServices;
2 using Platform.Numbers;
3 using static System.Runtime.CompilerServices.Unsafe;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.ResizableDirectMemory
8 {
9     public unsafe class LinksTargetsAVLBalancedTreeMethods<TLink> :
10         ↳ LinksAVLBalancedTreeMethodsBase<TLink>, ILinksTreeMethods<TLink>
11     {
12         public LinksTargetsAVLBalancedTreeMethods(ResizableDirectMemoryLinks<TLink> memory,
13             ↳ byte* links, byte* header) : base(memory, links, header) { }
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         protected unsafe override ref TLink GetLeftReference(TLink node) => ref
17             ↳ GetLinkReference(node).LeftAsTarget;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         protected unsafe override ref TLink GetRightReference(TLink node) => ref
21             ↳ GetLinkReference(node).RightAsTarget;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsTarget;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsTarget;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected override void SetLeft(TLink node, TLink left) =>
31             ↳ GetLinkReference(node).LeftAsTarget = left;
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected override void SetRight(TLink node, TLink right) =>
35             ↳ GetLinkReference(node).RightAsTarget = right;
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected override TLink GetSize(TLink node) =>
39             ↳ GetSizeValue(GetLinkReference(node).SizeAsTarget);
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
43             ↳ GetLinkReference(node).SizeAsTarget, size);
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected override bool GetLeftIsChild(TLink node) =>
47             ↳ GetLeftIsChildValue(GetLinkReference(node).SizeAsTarget);
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override void SetLeftIsChild(TLink node, bool value) =>
51             ↳ SetLeftIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         protected override bool GetRightIsChild(TLink node) =>
55             ↳ GetRightIsChildValue(GetLinkReference(node).SizeAsTarget);
56
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         protected override void SetRightIsChild(TLink node, bool value) =>
59             ↳ SetRightIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);
60
61         [MethodImpl(MethodImplOptions.AggressiveInlining)]
62         protected override sbyte GetBalance(TLink node) =>
63             ↳ GetBalanceValue(GetLinkReference(node).SizeAsTarget);
64
65         [MethodImpl(MethodImplOptions.AggressiveInlining)]
66         protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
67             ↳ GetLinkReference(node).SizeAsTarget, value);
68
69         [MethodImpl(MethodImplOptions.AggressiveInlining)]
70         protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsTarget;
71
72         [MethodImpl(MethodImplOptions.AggressiveInlining)]
73         protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
74
75         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```



```

62     protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
        ↪ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
        ↪ (IsEquals(firstTarget, secondTarget) && LessThan(firstSource, secondSource));
63
64     [MethodImpl(MethodImplOptions.AggressiveInlining)]
65     protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
        ↪ TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
        ↪ (IsEquals(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource));
66
67     [MethodImpl(MethodImplOptions.AggressiveInlining)]
68     protected override void ClearNode(TLink node)
69     {
70         ref var link = ref GetLinkReference(node);
71         link.LeftAsTarget = Zero;
72         link.RightAsTarget = Zero;
73         link.SizeAsTarget = Zero;
74     }
75 }
76 }

```

./Platform.Data.Doublets/ResizableDirectMemory/RawLink.cs

```

1  using Platform.Unsafe;
2
3  namespace Platform.Data.Doublets.ResizableDirectMemory
4  {
5      internal struct RawLink<TLink>
6      {
7          public static readonly long SizeInBytes = Structure<RawLink<TLink>>.Size;
8
9          public TLink Source;
10         public TLink Target;
11         public TLink LeftAsSource;
12         public TLink RightAsSource;
13         public TLink SizeAsSource;
14         public TLink LeftAsTarget;
15         public TLink RightAsTarget;
16         public TLink SizeAsTarget;
17     }
18 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Disposables;
5  using Platform.Singletons;
6  using Platform.Collections.Arrays;
7  using Platform.Numbers;
8  using Platform.Memory;
9  using Platform.Data.Exceptions;
10 using static Platform.Numbers.Arithmetic;
11 using static System.Runtime.CompilerServices.Unsafe;
12
13 #pragma warning disable 0649
14 #pragma warning disable 169
15 #pragma warning disable 618
16 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
17
18 // ReSharper disable StaticMemberInGenericType
19 // ReSharper disable BuiltInTypeReferenceStyle
20 // ReSharper disable MemberCanBePrivate.Local
21 // ReSharper disable UnusedMember.Local
22
23 namespace Platform.Data.Doublets.ResizableDirectMemory
24 {
25     public unsafe partial class ResizableDirectMemoryLinks<TLink> : DisposableBase, ILinks<TLink>
26     {
27         private static readonly EqualityComparer<TLink> _equalityComparer =
28             ↪ EqualityComparer<TLink>.Default;
29         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
30
31         /// <summary>Возвращает размер одной связи в байтах.</summary>
32         public static readonly long LinkSizeInBytes = RawLink<TLink>.SizeInBytes;
33
34         public static readonly long LinkHeaderSizeInBytes = LinksHeader<TLink>.SizeInBytes;
35
36         public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
37
38         private readonly long _memoryReservationStep;
39         private readonly IResizableDirectMemory _memory;

```

```

40 private byte* _header;
41 private byte* _links;
42
43 private ILinksTreeMethods<TLink> _targetsTreeMethods;
44 private ILinksTreeMethods<TLink> _sourcesTreeMethods;
45
46 // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
47 // ↳ нужно использовать не список а дерево, так как так можно быстрее проверить на
48 // ↳ наличие связи внутри
49 private ILinksListMethods<TLink> _unusedLinksListMethods;
50
51 /// <summary>
52 /// Возвращает общее число связей находящихся в хранилище.
53 /// </summary>
54 private TLink Total
55 {
56     get
57     {
58         ref var header = ref AsRef<LinksHeader<TLink>>(_header);
59         return Subtract(header.AllocatedLinks, header.FreeLinks);
60     }
61 }
62
63 public LinksConstants<TLink> Constants { get; }
64
65 public ResizableDirectMemoryLinks(string address) : this(address, DefaultLinksSizeStep)
66 ↳ { }
67
68 /// <summary>
69 /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
70 ↳ минимальным шагом расширения базы данных.
71 /// </summary>
72 /// <param name="address">Полный путь к файлу базы данных.</param>
73 /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
74 ↳ байтах.</param>
75 public ResizableDirectMemoryLinks(string address, long memoryReservationStep) : this(new
76 ↳ FileMappedResizableDirectMemory(address, memoryReservationStep),
77 ↳ memoryReservationStep) { }
78
79 public ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
80 ↳ DefaultLinksSizeStep) { }
81
82 public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
83 ↳ memoryReservationStep)
84 {
85     Constants = Default<LinksConstants<TLink>>.Instance;
86     _memory = memory;
87     _memoryReservationStep = memoryReservationStep;
88     if (memory.ReservedCapacity < memoryReservationStep)
89     {
90         memory.ReservedCapacity = memoryReservationStep;
91     }
92     SetPointers(_memory);
93     ref var header = ref AsRef<LinksHeader<TLink>>(_header);
94     // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
95     _memory.UsedCapacity = ((Integer<TLink>)header.AllocatedLinks * LinkSizeInBytes) +
96     ↳ LinkHeaderSizeInBytes;
97     // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
98     header.ReservedLinks = (Integer<TLink>)((_memory.ReservedCapacity -
99     ↳ LinkHeaderSizeInBytes) / LinkSizeInBytes);
100 }
101
102 [MethodImpl(MethodImplOptions.AggressiveInlining)]
103 public TLink Count(IList<TLink> restrictions)
104 {
105     // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
106     if (restrictions.Count == 0)
107     {
108         return Total;
109     }
110     if (restrictions.Count == 1)
111     {
112         var index = restrictions[Constants.IndexPart];
113         if (_equalityComparer.Equals(index, Constants.Any))
114         {
115             return Total;
116         }
117         return Exists(index) ? Integer<TLink>.One : Integer<TLink>.Zero;
118     }
119 }

```

```

108 if (restrictions.Count == 2)
109 {
110     var index = restrictions[Constants.IndexPart];
111     var value = restrictions[1];
112     if (_equalityComparer.Equals(index, Constants.Any))
113     {
114         if (_equalityComparer.Equals(value, Constants.Any))
115         {
116             return Total; // Any - как отсутствие ограничения
117         }
118         return Add(_sourcesTreeMethods.CountUsages(value),
119             ↪ _targetsTreeMethods.CountUsages(value));
120     }
121     else
122     {
123         if (!Exists(index))
124         {
125             return Integer<TLink>.Zero;
126         }
127         if (_equalityComparer.Equals(value, Constants.Any))
128         {
129             return Integer<TLink>.One;
130         }
131         ref var storedLinkValue = ref GetLinkUnsafe(index);
132         if (_equalityComparer.Equals(storedLinkValue.Source, value) ||
133             ↪ _equalityComparer.Equals(storedLinkValue.Target, value))
134         {
135             return Integer<TLink>.One;
136         }
137         return Integer<TLink>.Zero;
138     }
139 }
140 if (restrictions.Count == 3)
141 {
142     var index = restrictions[Constants.IndexPart];
143     var source = restrictions[Constants.SourcePart];
144     var target = restrictions[Constants.TargetPart];
145     if (_equalityComparer.Equals(index, Constants.Any))
146     {
147         if (_equalityComparer.Equals(source, Constants.Any) &&
148             ↪ _equalityComparer.Equals(target, Constants.Any))
149         {
150             return Total;
151         }
152         else if (_equalityComparer.Equals(source, Constants.Any))
153         {
154             return _targetsTreeMethods.CountUsages(target);
155         }
156         else if (_equalityComparer.Equals(target, Constants.Any))
157         {
158             return _sourcesTreeMethods.CountUsages(source);
159         }
160         else //if(source != Any && target != Any)
161         {
162             // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
163             var link = _sourcesTreeMethods.Search(source, target);
164             return _equalityComparer.Equals(link, Constants.Null) ?
165                 ↪ Integer<TLink>.Zero : Integer<TLink>.One;
166         }
167     }
168     else
169     {
170         if (!Exists(index))
171         {
172             return Integer<TLink>.Zero;
173         }
174         if (_equalityComparer.Equals(source, Constants.Any) &&
175             ↪ _equalityComparer.Equals(target, Constants.Any))
176         {
177             return Integer<TLink>.One;
178         }
179         ref var storedLinkValue = ref GetLinkUnsafe(index);
180         if (!_equalityComparer.Equals(source, Constants.Any) &&
181             ↪ !_equalityComparer.Equals(target, Constants.Any))
182         {
183             if (_equalityComparer.Equals(storedLinkValue.Source, source) &&
184                 ↪ _equalityComparer.Equals(storedLinkValue.Target, target))

```

```

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

```

```

254     }
255     if (_equalityComparer.Equals(value, Constants.Any))
256     {
257         return handler(GetLinkStruct(index));
258     }
259     ref var storedLinkValue = ref GetLinkUnsafe(index);
260     if (_equalityComparer.Equals(storedLinkValue.Source, value) ||
261         _equalityComparer.Equals(storedLinkValue.Target, value))
262     {
263         return handler(GetLinkStruct(index));
264     }
265     return Constants.Continue;
266 }
267
268 if (restrictions.Count == 3)
269 {
270     var index = restrictions[Constants.IndexPart];
271     var source = restrictions[Constants.SourcePart];
272     var target = restrictions[Constants.TargetPart];
273     if (_equalityComparer.Equals(index, Constants.Any))
274     {
275         if (_equalityComparer.Equals(source, Constants.Any) &&
276             ↪ _equalityComparer.Equals(target, Constants.Any))
277         {
278             return Each(handler, ArrayPool<TLink>.Empty);
279         }
280         else if (_equalityComparer.Equals(source, Constants.Any))
281         {
282             return _targetsTreeMethods.EachUsage(target, handler);
283         }
284         else if (_equalityComparer.Equals(target, Constants.Any))
285         {
286             return _sourcesTreeMethods.EachUsage(source, handler);
287         }
288         else //if(source != Any && target != Any)
289         {
290             var link = _sourcesTreeMethods.Search(source, target);
291             return _equalityComparer.Equals(link, Constants.Null) ?
292                 ↪ Constants.Continue : handler(GetLinkStruct(link));
293         }
294     }
295     else
296     {
297         if (!Exists(index))
298         {
299             return Constants.Continue;
300         }
301         if (_equalityComparer.Equals(source, Constants.Any) &&
302             ↪ _equalityComparer.Equals(target, Constants.Any))
303         {
304             return handler(GetLinkStruct(index));
305         }
306         ref var storedLinkValue = ref GetLinkUnsafe(index);
307         if (!_equalityComparer.Equals(source, Constants.Any) &&
308             ↪ !_equalityComparer.Equals(target, Constants.Any))
309         {
310             if (_equalityComparer.Equals(storedLinkValue.Source, source) &&
311                 _equalityComparer.Equals(storedLinkValue.Target, target))
312             {
313                 return handler(GetLinkStruct(index));
314             }
315             return Constants.Continue;
316         }
317         var value = default(TLink);
318         if (_equalityComparer.Equals(source, Constants.Any))
319         {
320             value = target;
321         }
322         if (_equalityComparer.Equals(target, Constants.Any))
323         {
324             value = source;
325         }
326         if (_equalityComparer.Equals(storedLinkValue.Source, value) ||
327             _equalityComparer.Equals(storedLinkValue.Target, value))
328         {
329             return handler(GetLinkStruct(index));
330         }
331         return Constants.Continue;
332     }
333 }

```

```

328     }
329 }
330 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↳ поддерживаются.");
331 }
332
333 /// <remarks>
334 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
    ↳ в другом месте (но не в менеджере памяти, а в логике Links)
335 /// </remarks>
336 [MethodImpl(MethodImplOptions.AggressiveInlining)]
337 public TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
338 {
339     var linkIndex = restrictions[Constants.IndexPart];
340     ref var link = ref GetLinkUnsafe(linkIndex);
341     ref var firstAsSource = ref AsRef<LinksHeader<TLink>>(_header).FirstAsSource;
342     ref var firstAsTarget = ref AsRef<LinksHeader<TLink>>(_header).FirstAsTarget;
343     // Будет корректно работать только в том случае, если пространство выделенной связи
    ↳ предварительно заполнено нулями
344     if (!_equalityComparer.Equals(link.Source, Constants.Null))
345     {
346         _sourcesTreeMethods.Detach(ref firstAsSource, linkIndex);
347     }
348     if (!_equalityComparer.Equals(link.Target, Constants.Null))
349     {
350         _targetsTreeMethods.Detach(ref firstAsTarget, linkIndex);
351     }
352     link.Source = substitution[Constants.SourcePart];
353     link.Target = substitution[Constants.TargetPart];
354     if (!_equalityComparer.Equals(link.Source, Constants.Null))
355     {
356         _sourcesTreeMethods.Attach(ref firstAsSource, linkIndex);
357     }
358     if (!_equalityComparer.Equals(link.Target, Constants.Null))
359     {
360         _targetsTreeMethods.Attach(ref firstAsTarget, linkIndex);
361     }
362     return linkIndex;
363 }
364
365 [MethodImpl(MethodImplOptions.AggressiveInlining)]
366 public Link<TLink> GetLinkStruct(TLink linkIndex)
367 {
368     ref var link = ref GetLinkUnsafe(linkIndex);
369     return new Link<TLink>(linkIndex, link.Source, link.Target);
370 }
371
372 [MethodImpl(MethodImplOptions.AggressiveInlining)]
373 internal ref RawLink<TLink> GetLinkUnsafe(TLink linkIndex) => ref
    ↳ AsRef<RawLink<TLink>>(_links + LinkSizeInBytes * (Integer<TLink>)linkIndex);
374
375 /// <remarks>
376 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
    ↳ пространство
377 /// </remarks>
378 public TLink Create(IList<TLink> restrictions)
379 {
380     ref var header = ref AsRef<LinksHeader<TLink>>(_header);
381     var freeLink = header.FirstFreeLink;
382     if (!_equalityComparer.Equals(freeLink, Constants.Null))
383     {
384         _unusedLinksListMethods.Detach(freeLink);
385     }
386     else
387     {
388         var maximumPossibleInnerReference =
            ↳ Constants.PossibleInnerReferencesRange.Maximum;
389         if (_comparer.Compare(header.AllocatedLinks, maximumPossibleInnerReference) > 0)
390         {
391             throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
392         }
393         if (_comparer.Compare(header.AllocatedLinks, Decrement(header.ReservedLinks)) >=
            ↳ 0)
394         {
395             _memory.ReservedCapacity += _memoryReservationStep;
396             SetPointers(_memory);
397             header.ReservedLinks = (Integer<TLink>)(_memory.ReservedCapacity /
            ↳ LinkSizeInBytes);

```

```

398     }
399     header.AllocatedLinks = Increment(header.AllocatedLinks);
400     _memory.UsedCapacity += LinkSizeInBytes;
401     freeLink = header.AllocatedLinks;
402 }
403 return freeLink;
404 }
405
406 public void Delete(IList<TLink> restrictions)
407 {
408     ref var header = ref AsRef<LinksHeader<TLink>>(_header);
409     var link = restrictions[Constants.IndexPart];
410     if (_comparer.Compare(link, header.AllocatedLinks) < 0)
411     {
412         _unusedLinksListMethods.AttachAsFirst(link);
413     }
414     else if (_equalityComparer.Equals(link, header.AllocatedLinks))
415     {
416         header.AllocatedLinks = Decrement(header.AllocatedLinks);
417         _memory.UsedCapacity -= LinkSizeInBytes;
418         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
419         //   ↳ пока не дойдём до первой существующей связи
420         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
421         while ((_comparer.Compare(header.AllocatedLinks, Integer<TLink>.Zero) > 0) &&
422             ↳ IsUnusedLink(header.AllocatedLinks))
423         {
424             _unusedLinksListMethods.Detach(header.AllocatedLinks);
425             header.AllocatedLinks = Decrement(header.AllocatedLinks);
426             _memory.UsedCapacity -= LinkSizeInBytes;
427         }
428     }
429 }
430
431 /// <remarks>
432 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
433   ↳ адрес реально поменялся
434 ///
435 /// Указатель this.links может быть в том же месте,
436 /// так как 0-я связь не используется и имеет такой же размер как Header,
437 /// поэтому header размещается в том же месте, что и 0-я связь
438 /// </remarks>
439 private void SetPointers(IDirectMemory memory)
440 {
441     if (memory == null)
442     {
443         _links = null;
444         _header = _links;
445         _unusedLinksListMethods = null;
446         _targetsTreeMethods = null;
447         _unusedLinksListMethods = null;
448     }
449     else
450     {
451         _links = (byte*)(void*)memory.Pointer;
452         _header = _links;
453         _sourcesTreeMethods = new LinksSourcesAVLBalancedTreeMethods<TLink>(this,
454             ↳ _links, _header);
455         _targetsTreeMethods = new LinksTargetsAVLBalancedTreeMethods<TLink>(this,
456             ↳ _links, _header);
457         _unusedLinksListMethods = new UnusedLinksListMethods<TLink>(_links, _header);
458     }
459 }
460
461 [MethodImpl(MethodImplOptions.AggressiveInlining)]
462 private bool Exists(TLink link)
463 => (_comparer.Compare(link, Constants.PossibleInnerReferencesRange.Minimum) >= 0)
464     && (_comparer.Compare(link, AsRef<LinksHeader<TLink>>(_header).AllocatedLinks) <= 0)
465     && !IsUnusedLink(link);
466
467 [MethodImpl(MethodImplOptions.AggressiveInlining)]
468 private bool IsUnusedLink(TLink link)
469 => _equalityComparer.Equals(AsRef<LinksHeader<TLink>>(_header).FirstFreeLink, link)
470     || (_equalityComparer.Equals(GetLinkUnsafe(link).SizeAsSource, Constants.Null)
471     && !_equalityComparer.Equals(GetLinkUnsafe(link).Source, Constants.Null));
472
473 #region DisposableBase
474
475 protected override bool AllowMultipleDisposeCalls => true;
476
477

```

```

472     protected override void Dispose(bool manual, bool wasDisposed)
473     {
474         if (!wasDisposed)
475         {
476             SetPointers(null);
477             _memory.DisposeIfPossible();
478         }
479     }
480
481     #endregion
482 }
483 }

```

./Platform.Data.Doublets/ResizableDirectMemory/UInt64LinksAVLBalancedTreeMethodsBase.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using System.Text;
5  using Platform.Collections.Methods.Trees;
6  using static System.Runtime.CompilerServices.Unsafe;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.ResizableDirectMemory
11 {
12     public unsafe abstract class UInt64LinksAVLBalancedTreeMethodsBase :
13         ↳ SizedAndThreadedAVLBalancedTreeMethods<ulong>
14     {
15         private readonly UInt64ResizableDirectMemoryLinks _memory;
16         private readonly LinksConstants<ulong> _constants;
17         internal readonly UInt64RawLink* _links;
18         internal readonly UInt64LinksHeader* _header;
19
20         internal UInt64LinksAVLBalancedTreeMethodsBase(UInt64ResizableDirectMemoryLinks memory,
21             ↳ UInt64RawLink* links, UInt64LinksHeader* header)
22         {
23             _links = links;
24             _header = header;
25             _memory = memory;
26             _constants = memory.Constants;
27
28             [MethodImpl(MethodImplOptions.AggressiveInlining)]
29             protected override ulong GetZero() => OUL;
30
31             [MethodImpl(MethodImplOptions.AggressiveInlining)]
32             protected override bool EqualToZero(ulong value) => value == OUL;
33
34             [MethodImpl(MethodImplOptions.AggressiveInlining)]
35             protected override bool IsEquals(ulong first, ulong second) => first == second;
36
37             [MethodImpl(MethodImplOptions.AggressiveInlining)]
38             protected override bool GreaterThanZero(ulong value) => value > OUL;
39
40             [MethodImpl(MethodImplOptions.AggressiveInlining)]
41             protected override bool GreaterThan(ulong first, ulong second) => first > second;
42
43             [MethodImpl(MethodImplOptions.AggressiveInlining)]
44             protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
45
46             [MethodImpl(MethodImplOptions.AggressiveInlining)]
47             protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
48                 ↳ always true for ulong
49
50             [MethodImpl(MethodImplOptions.AggressiveInlining)]
51             protected override bool LessOrEqualThanZero(ulong value) => value == OUL; // value is
52                 ↳ always >= 0 for ulong
53
54             [MethodImpl(MethodImplOptions.AggressiveInlining)]
55             protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
56
57             [MethodImpl(MethodImplOptions.AggressiveInlining)]
58             protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
59                 ↳ for ulong
60
61             [MethodImpl(MethodImplOptions.AggressiveInlining)]
62             protected override bool LessThan(ulong first, ulong second) => first < second;
63
64             [MethodImpl(MethodImplOptions.AggressiveInlining)]
65             protected override ulong Increment(ulong value) => ++value;
66         }
67     }
68 }

```



```

62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 protected override ulong Decrement(ulong value) => --value;
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 protected override ulong Add(ulong first, ulong second) => first + second;
67
68 [MethodImpl(MethodImplOptions.AggressiveInlining)]
69 protected override ulong Subtract(ulong first, ulong second) => first - second;
70
71 [MethodImpl(MethodImplOptions.AggressiveInlining)]
72 protected abstract ulong GetTreeRoot();
73
74 [MethodImpl(MethodImplOptions.AggressiveInlining)]
75 protected abstract ulong GetBasePartValue(ulong link);
76
77 [MethodImpl(MethodImplOptions.AggressiveInlining)]
78 protected abstract bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
79     ↳ ulong secondSource, ulong secondTarget);
80
81 [MethodImpl(MethodImplOptions.AggressiveInlining)]
82 protected abstract bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
83     ↳ ulong secondSource, ulong secondTarget);
84
85 [MethodImpl(MethodImplOptions.AggressiveInlining)]
86 protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
87 {
88     ref var firstLink = ref _links[first];
89     ref var secondLink = ref _links[second];
90     return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
91     ↳ secondLink.Source, secondLink.Target);
92 }
93
94 [MethodImpl(MethodImplOptions.AggressiveInlining)]
95 protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
96 {
97     ref var firstLink = ref _links[first];
98     ref var secondLink = ref _links[second];
99     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
100     ↳ secondLink.Source, secondLink.Target);
101 }
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 protected static ulong GetSizeValue(ulong value) => unchecked((value & 4294967264UL) >>
105     ↳ 5);
106
107 [MethodImpl(MethodImplOptions.AggressiveInlining)]
108 protected static void SetSizeValue(ref ulong storedValue, ulong size) => storedValue =
109     ↳ unchecked((storedValue & 31UL) | ((size & 134217727UL) << 5));
110
111 [MethodImpl(MethodImplOptions.AggressiveInlining)]
112 protected static bool GetLeftIsChildValue(ulong value) => unchecked((value & 16UL) >> 4
113     ↳ == 1UL);
114
115 [MethodImpl(MethodImplOptions.AggressiveInlining)]
116 protected static void SetLeftIsChildValue(ref ulong storedValue, bool value) =>
117     ↳ storedValue = unchecked((storedValue & 4294967279UL) | ((As<bool, byte>(ref value) &
118     ↳ 1UL) << 4));
119
120 [MethodImpl(MethodImplOptions.AggressiveInlining)]
121 protected static bool GetRightIsChildValue(ulong value) => unchecked((value & 8UL) >> 3
122     ↳ == 1UL);
123
124 [MethodImpl(MethodImplOptions.AggressiveInlining)]
125 protected static void SetRightIsChildValue(ref ulong storedValue, bool value) =>
126     ↳ storedValue = unchecked((storedValue & 4294967287UL) | ((As<bool, byte>(ref value) &
127     ↳ 1UL) << 3));
128
129 [MethodImpl(MethodImplOptions.AggressiveInlining)]
130 protected static sbyte GetBalanceValue(ulong value) => unchecked((sbyte)((value & 7UL) |
131     ↳ 0xF8UL * ((value & 4UL) >> 2))); // if negative, then continue ones to the end of
132     ↳ sbyte
133
134 [MethodImpl(MethodImplOptions.AggressiveInlining)]
135 protected static void SetBalanceValue(ref ulong storedValue, sbyte value) => storedValue
136     ↳ = unchecked((storedValue & 4294967288UL) | ((ulong)((byte)value >> 5) & 4) | value
137     ↳ & 3) & 7UL));

```

```

124 public ulong this[ulong index]
125 {
126     get
127     {
128         var root = GetTreeRoot();
129         if (index >= GetSize(root))
130         {
131             return OUL;
132         }
133         while (root != OUL)
134         {
135             var left = GetLeftOrDefault(root);
136             var leftSize = GetSizeOrZero(left);
137             if (index < leftSize)
138             {
139                 root = left;
140                 continue;
141             }
142             if (index == leftSize)
143             {
144                 return root;
145             }
146             root = GetRightOrDefault(root);
147             index -= leftSize + 1UL;
148         }
149         return OUL; // TODO: Impossible situation exception (only if tree structure
150                     ↳ broken)
151     }
152 }
153
154 /// <summary>
155 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
156 ↳ (концом).
157 /// </summary>
158 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
159 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
160 /// <returns>Индекс искомой связи.</returns>
161 public ulong Search(ulong source, ulong target)
162 {
163     var root = GetTreeRoot();
164     while (root != OUL)
165     {
166         var rootLink = _links[root];
167         var rootSource = rootLink.Source;
168         var rootTarget = rootLink.Target;
169         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
170             ↳ node.Key < root.Key
171         {
172             root = GetLeftOrDefault(root);
173         }
174         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
175             ↳ node.Key > root.Key
176         {
177             root = GetRightOrDefault(root);
178         }
179         else // node.Key == root.Key
180         {
181             return root;
182         }
183     }
184     return OUL;
185 }
186
187 // TODO: Return indices range instead of references count
188 public ulong CountUsages(ulong link)
189 {
190     var root = GetTreeRoot();
191     var total = GetSize(root);
192     var totalRightIgnore = OUL;
193     while (root != OUL)
194     {
195         var @base = GetBasePartValue(root);
196         if (@base <= link)
197         {
198             root = GetRightOrDefault(root);
199         }
200         else
201         {
202             totalRightIgnore += GetRightSize(root) + 1UL;
203         }
204     }
205     return total - totalRightIgnore;
206 }

```

```

199         root = GetLeftOrDefault(root);
200     }
201 }
202 root = GetTreeRoot();
203 var totalLeftIgnore = 0UL;
204 while (root != 0UL)
205 {
206     var @base = GetBasePartValue(root);
207     if (@base >= link)
208     {
209         root = GetLeftOrDefault(root);
210     }
211     else
212     {
213         totalLeftIgnore += GetLeftSize(root) + 1UL;
214         root = GetRightOrDefault(root);
215     }
216 }
217 return total - totalRightIgnore - totalLeftIgnore;
218 }
219
220 public ulong EachUsage(ulong link, Func<IList<ulong>, ulong> handler)
221 {
222     var root = GetTreeRoot();
223     if (root == 0UL)
224     {
225         return _constants.Continue;
226     }
227     ulong first = 0UL, current = root;
228     while (current != 0UL)
229     {
230         var @base = GetBasePartValue(current);
231         if (@base >= link)
232         {
233             if (@base == link)
234             {
235                 first = current;
236             }
237             current = GetLeftOrDefault(current);
238         }
239         else
240         {
241             current = GetRightOrDefault(current);
242         }
243     }
244     if (first != 0UL)
245     {
246         current = first;
247         while (true)
248         {
249             if (handler(_memory.GetLinkStruct(current)) == _constants.Break)
250             {
251                 return _constants.Break;
252             }
253             current = GetNext(current);
254             if (current == 0UL || GetBasePartValue(current) != link)
255             {
256                 break;
257             }
258         }
259     }
260     return _constants.Continue;
261 }
262
263 protected override void PrintNodeValue(ulong node, StringBuilder sb)
264 {
265     ref var link = ref _links[node];
266     sb.Append(' ');
267     sb.Append(link.Source);
268     sb.Append('-');
269     sb.Append('>');
270     sb.Append(link.Target);
271 }
272 }
273 }

```

./Platform.Data.Doublets/ResizableDirectMemory/UInt64LinksHeader.cs

```

1 namespace Platform.Data.Doublets.ResizableDirectMemory
2 {

```

```

3     internal struct UInt64LinksHeader
4     {
5         public ulong AllocatedLinks;
6         public ulong ReservedLinks;
7         public ulong FreeLinks;
8         public ulong FirstFreeLink;
9         public ulong FirstAsSource;
10        public ulong FirstAsTarget;
11        public ulong LastFreeLink;
12        public ulong Reserved8;
13    }
14 }

```

./Platform.Data.Doublets/ResizableDirectMemory/UInt64LinksSourcesAVLBalancedTreeMethods.cs

```

1 using System.Runtime.CompilerServices;
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 class UInt64LinksSourcesAVLBalancedTreeMethods :
8     ↪ UInt64LinksAVLBalancedTreeMethodsBase, ILinksTreeMethods<ulong>
9     {
10         internal UInt64LinksSourcesAVLBalancedTreeMethods(UInt64ResizableDirectMemoryLinks
11         ↪ memory, UInt64RawLink* links, UInt64LinksHeader* header) : base(memory, links,
12         ↪ header) { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref ulong GetLeftReference(ulong node) => ref
16         ↪ _links[node].LeftAsSource;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref ulong GetRightReference(ulong node) => ref
20         ↪ _links[node].RightAsSource;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override ulong GetLeft(ulong node) => _links[node].LeftAsSource;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override ulong GetRight(ulong node) => _links[node].RightAsSource;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void SetLeft(ulong node, ulong left) => _links[node].LeftAsSource =
30         ↪ left;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetRight(ulong node, ulong right) => _links[node].RightAsSource
34         ↪ = right;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override ulong GetSize(ulong node) => GetSizeValue(_links[node].SizeAsSource);
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
41         ↪ _links[node].SizeAsSource, size);
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override bool GetLeftIsChild(ulong node) =>
45         ↪ GetLeftIsChildValue(_links[node].SizeAsSource);
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected override void SetLeftIsChild(ulong node, bool value) =>
49         ↪ SetLeftIsChildValue(ref _links[node].SizeAsSource, value);
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         protected override bool GetRightIsChild(ulong node) =>
53         ↪ GetRightIsChildValue(_links[node].SizeAsSource);
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected override void SetRightIsChild(ulong node, bool value) =>
57         ↪ SetRightIsChildValue(ref _links[node].SizeAsSource, value);
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         protected override sbyte GetBalance(ulong node) =>
61         ↪ GetBalanceValue(_links[node].SizeAsSource);
62
63         [MethodImpl(MethodImplOptions.AggressiveInlining)]
64         protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
65         ↪ _links[node].SizeAsSource, value);
66     }
67 }

```

```

52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     protected override ulong GetTreeRoot() => _header->FirstAsSource;
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     protected override ulong GetBasePartValue(ulong link) => _links[link].Source;
57
58     [MethodImpl(MethodImplOptions.AggressiveInlining)]
59     protected override bool FirstIsToLeftOfSecond(ulong firstSource, ulong firstTarget,
60         ↪     ulong secondSource, ulong secondTarget)
61         => firstSource < secondSource || (firstSource == secondSource && firstTarget <
62             ↪     secondTarget);
63
64     [MethodImpl(MethodImplOptions.AggressiveInlining)]
65     protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
66         ↪     ulong secondSource, ulong secondTarget)
67         => firstSource > secondSource || (firstSource == secondSource && firstTarget >
68             ↪     secondTarget);
69
70     [MethodImpl(MethodImplOptions.AggressiveInlining)]
71     protected override void ClearNode(ulong node)
72     {
73         ref UInt64RawLink link = ref _links[node];
74         link.LeftAsSource = OUL;
75         link.RightAsSource = OUL;
76         link.SizeAsSource = OUL;
77     }
78 }
79
80 }

```

```

1  ./Platform.Data.Doublets/ResizableDirectMemory/UInt64LinksTargetsAVLBalancedTreeMethods.cs
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 class UInt64LinksTargetsAVLBalancedTreeMethods :
8          ↳ UInt64LinksAVLBalancedTreeMethodsBase, ILinksTreeMethods<ulong>
9      {
10         internal UInt64LinksTargetsAVLBalancedTreeMethods(UInt64ResizableDirectMemoryLinks
11             ↳ memory, UInt64RawLink* links, UInt64LinksHeader* header) : base(memory, links,
12             ↳ header) { }
13
14         //protected override IntPtr GetLeft(ulong node) => new IntPtr(&Links[node].LeftAsTarget);
15
16         //protected override IntPtr GetRight(ulong node) => new
17             ↳ IntPtr(&Links[node].RightAsTarget);
18
19         //protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;
20
21         //protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
22             ↳ left;
23
24         //protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget
25             ↳ = right;
26
27         //protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsTarget =
28             ↳ size;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected override ref ulong GetLeftReference(ulong node) => ref
32             ↳ _links[node].LeftAsTarget;
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected override ref ulong GetRightReference(ulong node) => ref
36             ↳ _links[node].RightAsTarget;
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         protected override ulong GetLeft(ulong node) => _links[node].LeftAsTarget;
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         protected override ulong GetRight(ulong node) => _links[node].RightAsTarget;
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         protected override void SetLeft(ulong node, ulong left) => _links[node].LeftAsTarget =
46             ↳ left;
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         protected override void SetRight(ulong node, ulong right) => _links[node].RightAsTarget =
50             ↳ right;
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected override void SetSize(ulong node, ulong size) => _links[node].SizeAsTarget =
54             ↳ size;
55     }
56 }

```

```

39     protected override void SetRight(ulong node, ulong right) => _links[node].RightAsTarget
    ↪     = right;
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     protected override ulong GetSize(ulong node) => GetSizeValue(_links[node].SizeAsTarget);
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
    ↪     _links[node].SizeAsTarget, size);
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     protected override bool GetLeftIsChild(ulong node) =>
    ↪     GetLeftIsChildValue(_links[node].SizeAsTarget);
49
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     protected override void SetLeftIsChild(ulong node, bool value) =>
    ↪     SetLeftIsChildValue(ref _links[node].SizeAsTarget, value);
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected override bool GetRightIsChild(ulong node) =>
    ↪     GetRightIsChildValue(_links[node].SizeAsTarget);
55
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     protected override void SetRightIsChild(ulong node, bool value) =>
    ↪     SetRightIsChildValue(ref _links[node].SizeAsTarget, value);
58
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     protected override sbyte GetBalance(ulong node) =>
    ↪     GetBalanceValue(_links[node].SizeAsTarget);
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
    ↪     _links[node].SizeAsTarget, value);
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     protected override ulong GetTreeRoot() => _header->FirstAsTarget;
67
68     [MethodImpl(MethodImplOptions.AggressiveInlining)]
69     protected override ulong GetBasePartValue(ulong link) => _links[link].Target;
70
71     [MethodImpl(MethodImplOptions.AggressiveInlining)]
72     protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
    ↪     ulong secondSource, ulong secondTarget)
73     => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
    ↪     secondSource);
74
75     [MethodImpl(MethodImplOptions.AggressiveInlining)]
76     protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
    ↪     ulong secondSource, ulong secondTarget)
77     => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
    ↪     secondSource);
78
79     [MethodImpl(MethodImplOptions.AggressiveInlining)]
80     protected override void ClearNode(ulong node)
81     {
82         ref UInt64RawLink link = ref _links[node];
83         link.LeftAsTarget = OUL;
84         link.RightAsTarget = OUL;
85         link.SizeAsTarget = OUL;
86     }
87 }
88 }

```

./Platform.Data.Doublets/ResizableDirectMemory/UInt64RawLink.cs

```

1  namespace Platform.Data.Doublets.ResizableDirectMemory
2  {
3      internal struct UInt64RawLink
4      {
5          public ulong Source;
6          public ulong Target;
7          public ulong LeftAsSource;
8          public ulong RightAsSource;
9          public ulong SizeAsSource;
10         public ulong LeftAsTarget;
11         public ulong RightAsTarget;
12         public ulong SizeAsTarget;
13     }
14 }

```

./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 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(UInt64RawLink);
30
31         public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
32
33         private readonly long _memoryReservationStep;
34
35         private readonly IResizableDirectMemory _memory;
36         private UInt64LinksHeader* _header;
37         private UInt64RawLink* _links;
38
39         private ILinksTreeMethods<id> _targetsTreeMethods;
40         private ILinksTreeMethods<id> _sourcesTreeMethods;
41
42         // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
43         // → нужно использовать не список а дерево, так как так можно быстрее проверить на
44         // → наличие связи внутри
45         private ILinksListMethods<id> _unusedLinksListMethods;
46
47         /// <summary>
48         /// Возвращает общее число связей находящихся в хранилище.
49         /// </summary>
50         private id Total => _header->AllocatedLinks - _header->FreeLinks;
51
52         // TODO: Дать возможность переопределять в конструкторе
53         public LinksConstants<id> Constants { get; }
54
55         public UInt64ResizableDirectMemoryLinks(string address) : this(address,
56             → DefaultLinksSizeStep) { }
57
58         /// <summary>
59         /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
60         /// → минимальным шагом расширения базы данных.
61         /// </summary>
62         /// <param name="address">Полный путь к файлу базы данных.</param>
63         /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
64         /// → байтах.</param>
65         public UInt64ResizableDirectMemoryLinks(string address, long memoryReservationStep) :
66             → this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
67             → memoryReservationStep) { }
68
69         public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
70             → DefaultLinksSizeStep) { }
71
72         public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
73             → memoryReservationStep)
74         {
75             Constants = Default<LinksConstants<id>>.Instance;
76             _memory = memory;
77             _memoryReservationStep = memoryReservationStep;
78             if (memory.ReservedCapacity < memoryReservationStep)
79             {
80                 memory.ReservedCapacity = memoryReservationStep;
81             }
82         }
83     }
84 }
```

```

72     }
73     SetPointers(_memory);
74     // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
75     _memory.UsedCapacity = ((long)_header->AllocatedLinks * sizeof(UInt64RawLink)) +
        ↳ sizeof(UInt64LinksHeader);
76     // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
77     _header->ReservedLinks = (id)((_memory.ReservedCapacity - sizeof(UInt64LinksHeader))
        ↳ / sizeof(UInt64RawLink));
78 }
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 public id Count(IList<id> restrictions)
82 {
83     // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
84     if (restrictions.Count == 0)
85     {
86         return Total;
87     }
88     if (restrictions.Count == 1)
89     {
90         var index = restrictions[Constants.IndexPart];
91         if (index == Constants.Any)
92         {
93             return Total;
94         }
95         return Exists(index) ? 1UL : 0UL;
96     }
97     if (restrictions.Count == 2)
98     {
99         var index = restrictions[Constants.IndexPart];
100         var value = restrictions[1];
101         if (index == Constants.Any)
102         {
103             if (value == Constants.Any)
104             {
105                 return Total; // Any - как отсутствие ограничения
106             }
107             return _sourcesTreeMethods.CountUsages(value)
108                 + _targetsTreeMethods.CountUsages(value);
109         }
110         else
111         {
112             if (!Exists(index))
113             {
114                 return 0;
115             }
116             if (value == Constants.Any)
117             {
118                 return 1;
119             }
120             var storedLinkValue = GetLinkUnsafe(index);
121             if (storedLinkValue->Source == value ||
122                 storedLinkValue->Target == value)
123             {
124                 return 1;
125             }
126             return 0;
127         }
128     }
129     if (restrictions.Count == 3)
130     {
131         var index = restrictions[Constants.IndexPart];
132         var source = restrictions[Constants.SourcePart];
133         var target = restrictions[Constants.TargetPart];
134         if (index == Constants.Any)
135         {
136             if (source == Constants.Any && target == Constants.Any)
137             {
138                 return Total;
139             }
140             else if (source == Constants.Any)
141             {
142                 return _targetsTreeMethods.CountUsages(target);
143             }
144             else if (target == Constants.Any)
145             {
146                 return _sourcesTreeMethods.CountUsages(source);
147             }

```



```

148     else //if(source != Any && target != Any)
149     {
150         // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
151         var link = _sourcesTreeMethods.Search(source, target);
152         return link == Constants.Null ? OUL : 1UL;
153     }
154 }
155 else
156 {
157     if (!Exists(index))
158     {
159         return 0;
160     }
161     if (source == Constants.Any && target == Constants.Any)
162     {
163         return 1;
164     }
165     var storedLinkValue = GetLinkUnsafe(index);
166     if (source != Constants.Any && target != Constants.Any)
167     {
168         if (storedLinkValue->Source == source &&
169             storedLinkValue->Target == target)
170         {
171             return 1;
172         }
173         return 0;
174     }
175     var value = default(id);
176     if (source == Constants.Any)
177     {
178         value = target;
179     }
180     if (target == Constants.Any)
181     {
182         value = source;
183     }
184     if (storedLinkValue->Source == value ||
185         storedLinkValue->Target == value)
186     {
187         return 1;
188     }
189     return 0;
190 }
191 }
192 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↳ поддерживаются.");
193 }
194
195 [MethodImpl(MethodImplOptions.AggressiveInlining)]
196 public id Each(Func<IList<id>, id> handler, IList<id> restrictions)
197 {
198     if (restrictions.Count == 0)
199     {
200         for (id link = 1; link <= _header->AllocatedLinks; link++)
201         {
202             if (Exists(link))
203             {
204                 if (handler(GetLinkStruct(link)) == Constants.Break)
205                 {
206                     return Constants.Break;
207                 }
208             }
209         }
210         return Constants.Continue;
211     }
212     if (restrictions.Count == 1)
213     {
214         var index = restrictions[Constants.IndexPart];
215         if (index == Constants.Any)
216         {
217             return Each(handler, ArrayPool<ulong>.Empty);
218         }
219         if (!Exists(index))
220         {
221             return Constants.Continue;
222         }
223         return handler(GetLinkStruct(index));
224     }
225     if (restrictions.Count == 2)

```

```

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

```

```

303         return Constants.Continue;
304     }
305     var value = default(id);
306     if (source == Constants.Any)
307     {
308         value = target;
309     }
310     if (target == Constants.Any)
311     {
312         value = source;
313     }
314     if (storedLinkValue->Source == value ||
315         storedLinkValue->Target == value)
316     {
317         return handler(GetLinkStruct(index));
318     }
319     return Constants.Continue;
320 }
321 }
322 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↳ поддерживаются.");
323 }
324
325 /// <remarks>
326 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
    ↳ в другом месте (но не в менеджере памяти, а в логике Links)
327 /// </remarks>
328 [MethodImpl(MethodImplOptions.AggressiveInlining)]
329 public id Update(IList<id> restrictions, IList<id> substitution)
330 {
331     var linkIndex = restrictions[Constants.IndexPart];
332     var link = GetLinkUnsafe(linkIndex);
333     // Будет корректно работать только в том случае, если пространство выделенной связи
    ↳ предварительно заполнено нулями
334     if (link->Source != Constants.Null)
335     {
336         _sourcesTreeMethods.Detach(ref _header->FirstAsSource, linkIndex);
337     }
338     if (link->Target != Constants.Null)
339     {
340         _targetsTreeMethods.Detach(ref _header->FirstAsTarget, linkIndex);
341     }
342     #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
343     var leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
344     var rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
345     if (leftTreeSize != rightTreeSize)
346     {
347         throw new Exception("One of the trees is broken.");
348     }
349     #endif
350     link->Source = substitution[Constants.SourcePart];
351     link->Target = substitution[Constants.TargetPart];
352     if (link->Source != Constants.Null)
353     {
354         _sourcesTreeMethods.Attach(ref _header->FirstAsSource, linkIndex);
355     }
356     if (link->Target != Constants.Null)
357     {
358         _targetsTreeMethods.Attach(ref _header->FirstAsTarget, linkIndex);
359     }
360     #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
361     leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
362     rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
363     if (leftTreeSize != rightTreeSize)
364     {
365         throw new Exception("One of the trees is broken.");
366     }
367     #endif
368     return linkIndex;
369 }
370
371 [MethodImpl(MethodImplOptions.AggressiveInlining)]
372 public IList<id> GetLinkStruct(id linkIndex)
373 {
374     var link = GetLinkUnsafe(linkIndex);
375     return new UInt64Link(linkIndex, link->Source, link->Target);
376 }
377

```

```

378 [MethodImpl(MethodImplOptions.AggressiveInlining)]
379 internal UInt64RawLink* GetLinkUnsafe(id linkIndex) => &_links[linkIndex];
380
381 /// <remarks>
382 /// TODO0: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
383 /// ↳ пространство
384 /// </remarks>
385 public id Create(IList<id> restrictions)
386 {
387     var freeLink = _header->FirstFreeLink;
388     if (freeLink != Constants.Null)
389     {
390         _unusedLinksListMethods.Detach(freeLink);
391     }
392     else
393     {
394         var maximumPossibleInnerReference =
395             ↳ Constants.PossibleInnerReferencesRange.Maximum;
396         if (_header->AllocatedLinks > maximumPossibleInnerReference)
397         {
398             throw new LinksLimitReachedException<id>(maximumPossibleInnerReference);
399         }
400         if (_header->AllocatedLinks >= _header->ReservedLinks - 1)
401         {
402             _memory.ReservedCapacity += _memory.ReservationStep;
403             SetPointers(_memory);
404             _header->ReservedLinks = (id)(_memory.ReservedCapacity /
405                 ↳ sizeof(UInt64RawLink));
406         }
407         _header->AllocatedLinks++;
408         _memory.UsedCapacity += sizeof(UInt64RawLink);
409         freeLink = _header->AllocatedLinks;
410     }
411     return freeLink;
412 }
413
414 public void Delete(IList<id> restrictions)
415 {
416     var link = restrictions[Constants.IndexPart];
417     if (link < _header->AllocatedLinks)
418     {
419         _unusedLinksListMethods.AttachAsFirst(link);
420     }
421     else if (link == _header->AllocatedLinks)
422     {
423         _header->AllocatedLinks--;
424         _memory.UsedCapacity -= sizeof(UInt64RawLink);
425         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
426         // ↳ пока не дойдём до первой существующей связи
427         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
428         while (_header->AllocatedLinks > 0 && IsUnusedLink(_header->AllocatedLinks))
429         {
430             _unusedLinksListMethods.Detach(_header->AllocatedLinks);
431             _header->AllocatedLinks--;
432             _memory.UsedCapacity -= sizeof(UInt64RawLink);
433         }
434     }
435 }
436
437 /// <remarks>
438 /// TODO0: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
439 /// ↳ адрес реально поменялся
440 ///
441 /// Указатель this.links может быть в том же месте,
442 /// так как 0-я связь не используется и имеет такой же размер как Header,
443 /// поэтому header размещается в том же месте, что и 0-я связь
444 /// </remarks>
445 private void SetPointers(IResizableDirectMemory memory)
446 {
447     if (memory == null)
448     {
449         _header = null;
450         _links = null;
451         _unusedLinksListMethods = null;
452         _targetsTreeMethods = null;
453         _unusedLinksListMethods = null;
454     }
455     else
456     {

```

```

452     _header = (UInt64LinksHeader*)(void*)memory.Pointer;
453     _links = (UInt64RawLink*)(void*)memory.Pointer;
454     _sourcesTreeMethods = new UInt64LinksSourcesAVLBalancedTreeMethods(this, _links,
455         ↪ _header);
456     _targetsTreeMethods = new UInt64LinksTargetsAVLBalancedTreeMethods(this, _links,
457         ↪ _header);
458     _unusedLinksListMethods = new UInt64UnusedLinksListMethods(_links, _header);
459 }
460 [MethodImpl(MethodImplOptions.AggressiveInlining)]
461 private bool Exists(id link) => link >= Constants.PossibleInnerReferencesRange.Minimum
462     ↪ && link <= _header->AllocatedLinks && !IsUnusedLink(link);
463 [MethodImpl(MethodImplOptions.AggressiveInlining)]
464 private bool IsUnusedLink(id link) => _header->FirstFreeLink == link
465     || (_links[link].SizeAsSource == Constants.Null &&
466         ↪ _links[link].Source != Constants.Null);
467 #region Disposable
468 protected override bool AllowMultipleDisposeCalls => true;
469 protected override void Dispose(bool manual, bool wasDisposed)
470 {
471     if (!wasDisposed)
472     {
473         SetPointers(null);
474         _memory.DisposeIfPossible();
475     }
476 }
477 #endregion
478 }
479 }
480 }
481 }
482 }

```

./Platform.Data.Doublets/ResizableDirectMemory/UInt64UnusedLinksListMethods.cs

```

1  using Platform.Collections.Methods.Lists;
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.ResizableDirectMemory
7  {
8      public unsafe class UInt64UnusedLinksListMethods : CircularDoublyLinkedListMethods<ulong>,
9          ↪ ILinksListMethods<ulong>
10     {
11         private readonly UInt64RawLink* _links;
12         private readonly UInt64LinksHeader* _header;
13
14         internal UInt64UnusedLinksListMethods(UInt64RawLink* links, UInt64LinksHeader* header)
15         {
16             _links = links;
17             _header = header;
18         }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected override ulong GetFirst() => _header->FirstFreeLink;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override ulong GetLast() => _header->LastFreeLink;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override ulong GetPrevious(ulong element) => _links[element].Source;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected override ulong GetNext(ulong element) => _links[element].Target;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override ulong GetSize() => _header->FreeLinks;
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override void SetFirst(ulong element) => _header->FirstFreeLink = element;
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         protected override void SetLast(ulong element) => _header->LastFreeLink = element;
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         protected override void SetPrevious(ulong element, ulong previous) =>
43             ↪ _links[element].Source = previous;

```

```

42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     protected override void SetNext(ulong element, ulong next) => _links[element].Target =
44         next;
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected override void SetSize(ulong size) => _header->FreeLinks = size;
48 }
49 }

```

./Platform.Data.Doublets/ResizableDirectMemory/UnusedLinksListMethods.cs

```

1  using Platform.Collections.Methods.Lists;
2  using Platform.Numbers;
3  using System.Runtime.CompilerServices;
4  using static System.Runtime.CompilerServices.Unsafe;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.ResizableDirectMemory
9  {
10     public unsafe class UnusedLinksListMethods<TLink> : CircularDoublyLinkedListMethods<TLink>,
11         ↳ ILinksListMethods<TLink>
12     {
13         private readonly byte* _links;
14         private readonly byte* _header;
15
16         public UnusedLinksListMethods(byte* links, byte* header)
17         {
18             _links = links;
19             _header = header;
20         }
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         internal virtual ref LinksHeader<TLink> GetHeader() => ref
24             ↳ AsRef<LinksHeader<TLink>>(_header);
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         internal virtual ref RawLink<TLink> GetLink(TLink link) => ref
28             ↳ AsRef<RawLink<TLink>>((void*)(_links + RawLink<TLink>.SizeInBytes *
29                 ↳ (Integer<TLink>)link));
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override TLink GetFirst() => GetHeader().FirstFreeLink;
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected override TLink GetLast() => GetHeader().LastFreeLink;
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected override TLink GetPrevious(TLink element) => GetLink(element).Source;
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected override TLink GetNext(TLink element) => GetLink(element).Target;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override TLink GetSize() => GetHeader().FreeLinks;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected override void SetFirst(TLink element) => GetHeader().FirstFreeLink = element;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override void SetLast(TLink element) => GetHeader().LastFreeLink = element;
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected override void SetPrevious(TLink element, TLink previous) =>
54             ↳ GetLink(element).Source = previous;
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         protected override void SetNext(TLink element, TLink next) => GetLink(element).Target =
58             ↳ next;
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         protected override void SetSize(TLink size) => GetHeader().FreeLinks = size;
62     }
63 }

```

./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(IList<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         private void HalveSequence(IList<TLink> destination, IList<TLink> source, int length)
41         {
42             var loopedLength = length - (length % 2);
43             for (var i = 0; i < loopedLength; i += 2)
44             {
45                 destination[i / 2] = Links.GetOrCreate(source[i], source[i + 1]);
46             }
47             if (length > loopedLength)
48             {
49                 destination[length / 2] = source[length - 1];
50             }
51         }
52     }
53 }

```

```

./Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs
1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Interfaces;
5 using Platform.Collections;
6 using Platform.Singletons;
7 using Platform.Numbers;
8 using Platform.Data.Doublets.Sequences.Frequencies.Cache;

```

```

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

```



```

77 {
78     if (sequence.IsNullOrEmpty())
79     {
80         return null;
81     }
82     if (sequence.Count == 1)
83     {
84         return sequence;
85     }
86     if (sequence.Count == 2)
87     {
88         return new[] { Links.GetOrCreate(sequence[0], sequence[1]) };
89     }
90     // TODO: arraypool with min size (to improve cache locality) or stackalloc with Sigil
91     var copy = new HalfDoublet[sequence.Count];
92     Doublet<TLink> doublet = default;
93     for (var i = 1; i < sequence.Count; i++)
94     {
95         doublet.Source = sequence[i - 1];
96         doublet.Target = sequence[i];
97         LinkFrequency<TLink> data;
98         if (_doInitialFrequenciesIncrement)
99         {
100             data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
101         }
102         else
103         {
104             data = _doubletFrequenciesCache.GetFrequency(ref doublet);
105             if (data == null)
106             {
107                 throw new NotSupportedException("If you ask not to increment
108                     ↪ frequencies, it is expected that all frequencies for the sequence
109                     ↪ are prepared.");
110             }
111             copy[i - 1].Element = sequence[i - 1];
112             copy[i - 1].DoubletData = data;
113             UpdateMaxDoublet(ref doublet, data);
114         }
115     }
116     copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
117     copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
118     if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
119     {
120         var newLength = ReplaceDoublets(copy);
121         sequence = new TLink[newLength];
122         for (int i = 0; i < newLength; i++)
123         {
124             sequence[i] = copy[i].Element;
125         }
126     }
127     return sequence;
128 }
129
130 /// <remarks>
131 /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding
132 /// </remarks>
133 private int ReplaceDoublets(HalfDoublet[] copy)
134 {
135     var oldLength = copy.Length;
136     var newLength = copy.Length;
137     while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
138     {
139         var maxDoubletSource = _maxDoublet.Source;
140         var maxDoubletTarget = _maxDoublet.Target;
141         if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
142         {
143             _maxDoubletData.Link = Links.GetOrCreate(maxDoubletSource, maxDoubletTarget);
144         }
145         var maxDoubletReplacementLink = _maxDoubletData.Link;
146         oldLength--;
147         var oldLengthMinusTwo = oldLength - 1;
148         // Substitute all usages
149         int w = 0, r = 0; // (r == read, w == write)
150         for (; r < oldLength; r++)
151         {
152             if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
153                 ↪ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
154             {
155                 if (r > 0)

```

```

153         {
154             var previous = copy[w - 1].Element;
155             copy[w - 1].DoubletData.DecrementFrequency();
156             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,
                ↳ next);
163         }
164         copy[w++].Element = maxDoubletReplacementLink;
165         r++;
166         newLength--;
167     }
168     else
169     {
170         copy[w++] = copy[r];
171     }
172 }
173 if (w < newLength)
174 {
175     copy[w] = copy[r];
176 }
177 oldLength = newLength;
178 ResetMaxDoublet();
179 UpdateMaxDoublet(copy, newLength);
180 }
181 return newLength;
182 }
183
184 [MethodImpl(MethodImplOptions.AggressiveInlining)]
185 private void ResetMaxDoublet()
186 {
187     _maxDoublet = new Doublet<TLink>();
188     _maxDoubletData = new LinkFrequency<TLink>();
189 }
190
191 [MethodImpl(MethodImplOptions.AggressiveInlining)]
192 private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
193 {
194     Doublet<TLink> doublet = default;
195     for (var i = 1; i < length; i++)
196     {
197         doublet.Source = copy[i - 1].Element;
198         doublet.Target = copy[i].Element;
199         UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
200     }
201 }
202
203 [MethodImpl(MethodImplOptions.AggressiveInlining)]
204 private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
205 {
206     var frequency = data.Frequency;
207     var maxFrequency = _maxDoubletData.Frequency;
208     //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
209     ↳ (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
210     ↳ compression string data (and gives collisions quickly) */ _maxDoublet.Source +
211     ↳ _maxDoublet.Target)))
212     if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
213         _comparer.Compare(maxFrequency, frequency) < 0 ||
214         (
215             ↳ _equalityComparer.Equals(maxFrequency, frequency) &&
216             ↳ _comparer.Compare(Arithmetic.Add(doublet.Source, doublet.Target),
217             ↳ Arithmetic.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /* gives
218     ↳ better stability and better compression on sequent data and even on random
219     ↳ numbers data (but gives collisions anyway) */
220     {
221         _maxDoublet = doublet;
222         _maxDoubletData = data;
223     }
224 }
225 }
226 }
227 }

```

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

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

./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                         }
64                     }
65                 }
66             }
67         }
```

```

58         skipOnce = true;
59     }
60     else if (i == length - 1)
61     {
62         sequence[w] = sequence[i];
63         levels[w] = levels[i];
64         w++;
65     }
66 }
67 else
68 {
69     currentLevel = levels[i];
70     levelRepeat = 1;
71     if (skipOnce)
72     {
73         skipOnce = false;
74     }
75     else
76     {
77         sequence[w] = sequence[i - 1];
78         levels[w] = levels[i - 1];
79         previousLevel = levels[w];
80         w++;
81     }
82     if (i == length - 1)
83     {
84         sequence[w] = sequence[i];
85         levels[w] = levels[i];
86         w++;
87     }
88 }
89 }
90 length = w;
91 }
92 return links.GetOrCreate(sequence[0], sequence[1]);
93 }
94
95 private static TLink GetGreatestNeighbourLowerThanCurrentOrCurrent(TLink previous, TLink
↪ 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],
27         ↪ sequence[sequence.Count - 1]);
28     return levels;
29 }
30 public TLink GetFrequencyNumber(TLink source, TLink target) =>
31     ↪ _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLink>(source, target));
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>,
8         ↪ ICriterionMatcher<TLink>
9     {
10         public DefaultSequenceElementCriterionMatcher(ILinks<TLink> links) : base(links) { }
11         public bool IsMatched(TLink argument) => Links.IsPartialPoint(argument);
12     }
13 }

```

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

```

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

```

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

```

1 using System.Collections.Generic;
2 using Platform.Collections.Stacks;
3 using Platform.Data.Doublets.Sequences.HeightProviders;
4 using Platform.Data.Sequences;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Sequences
9 {
10     public class DefaultSequenceAppender<TLink> : LinksOperatorBase<TLink>,
11         ↪ ISequenceAppender<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ↪ EqualityComparer<TLink>.Default;
15
16         private readonly IStack<TLink> _stack;
17         private readonly ISequenceHeightProvider<TLink> _heightProvider;
18
19         public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
20             ↪ ISequenceHeightProvider<TLink> heightProvider)
21             : base(links)
22         { }
23     }
24 }

```

```

19     {
20         _stack = stack;
21         _heightProvider = heightProvider;
22     }
23
24     public TLink Append(TLink sequence, TLink appendant)
25     {
26         var cursor = sequence;
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         }
42         var left = cursor;
43         var right = appendant;
44         while (!_equalityComparer.Equals(cursor = _stack.Pop(), Links.Constants.Null))
45         {
46             right = Links.GetOrCreate(left, right);
47             left = cursor;
48         }
49         return Links.GetOrCreate(left, right);
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 }

```

```

24 private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
    ↳ IList<TLink>>>
25 {
26     private readonly IListEqualityComparer<TLink> _listComparer;
27     public ItemEquilityComparer() => _listComparer =
    ↳ 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 }

```



```

3 using System.Runtime.CompilerServices;
4 using Platform.Interfaces;
5 using Platform.Numbers;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
10 {
11     /// <remarks>
12     /// Can be used to operate with many CompressingConverters (to keep global frequencies data
13     /// ↪ between them).
14     /// TODO: Extract interface to implement frequencies storage inside Links storage
15     /// </remarks>
16     public class LinkFrequenciesCache<TLink> : LinksOperatorBase<TLink>
17     {
18         private static readonly EqualityComparer<TLink> _equalityComparer =
19             ↪ EqualityComparer<TLink>.Default;
20         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
21
22         private readonly Dictionary<Doublet<TLink>, LinkFrequency<TLink>> _doubletsCache;
23         private readonly ICounter<TLink, TLink> _frequencyCounter;
24
25         public LinkFrequenciesCache(ILinks<TLink> links, ICounter<TLink, TLink> frequencyCounter)
26             : base(links)
27         {
28             _doubletsCache = new Dictionary<Doublet<TLink>, LinkFrequency<TLink>>(4096,
29                 ↪ DoubletComparer<TLink>.Default);
30             _frequencyCounter = frequencyCounter;
31         }
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
35         {
36             var doublet = new Doublet<TLink>(source, target);
37             return GetFrequency(ref doublet);
38         }
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
42         {
43             _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
44             return data;
45         }
46
47         public void IncrementFrequencies(IList<TLink> sequence)
48         {
49             for (var i = 1; i < sequence.Count; i++)
50             {
51                 IncrementFrequency(sequence[i - 1], sequence[i]);
52             }
53         }
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
57         {
58             var doublet = new Doublet<TLink>(source, target);
59             return IncrementFrequency(ref doublet);
60         }
61
62         public void PrintFrequencies(IList<TLink> sequence)
63         {
64             for (var i = 1; i < sequence.Count; i++)
65             {
66                 PrintFrequency(sequence[i - 1], sequence[i]);
67             }
68         }
69
70         public void PrintFrequency(TLink source, TLink target)
71         {
72             var number = GetFrequency(source, target).Frequency;
73             Console.WriteLine("{0},{1}) - {2}", source, target, number);
74         }
75
76         [MethodImpl(MethodImplOptions.AggressiveInlining)]
77         public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
78         {
79             if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
80             {
81                 data.IncrementFrequency();
82             }
83         }
84     }
85 }

```

```

79     }
80     else
81     {
82         var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
83         data = new LinkFrequency<TLink>(Integer<TLink>.One, link);
84         if (!_equalityComparer.Equals(link, default))
85         {
86             data.Frequency = Arithmetic.Add(data.Frequency,
87                 ↪ _frequencyCounter.Count(link));
88         }
89         _doubletsCache.Add(doublet, data);
90     }
91     return data;
92 }
93 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 }

```

./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 }

```

```

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

```

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

```

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

```

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

```

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

```

```

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

```

```

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

```

```

62         CountCore(doublenetIndex);
63     }
64     return constants.Continue;
65 }
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 }

```

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

```

1  using Platform.Interfaces;
2  using Platform.Numbers;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.HeightProviders
7  {
8      public class DefaultSequenceRightHeightProvider<TLink> : LinksOperatorBase<TLink>,
9          ↳ ISequenceHeightProvider<TLink>
10     {
11         private readonly ICriterionMatcher<TLink> _elementMatcher;
12
13         public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriterionMatcher<TLink>
14             ↳ elementMatcher) : base(links) => _elementMatcher = elementMatcher;
15
16         public TLink Get(TLink sequence)
17         {
18
19         }
20     }
21 }

```

```

16         var height = default(TLink);
17         var pairOrElement = sequence;
18         while (!_elementMatcher.IsMatched(pairOrElement))
19         {
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             // Implementation
21         }
22     }
23 }

```

```

17     {
18         var indexed = true;
19         var i = sequence.Count;
20         while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
21             ↳ { }
22         for (; i >= 1; i--)
23         {
24             _cache.IncrementFrequency(sequence[i - 1], sequence[i]);
25         }
26         return indexed;
27     }
28     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     public bool MightContain(ICollection<TLink> sequence)
43     {
44         var indexed = true;
45         var i = sequence.Count;
46         while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
47         return indexed;
48     }
49     private bool IsIndexed(TLink source, TLink target)
50     {
51         var frequency = _cache.GetFrequency(source, target);
52         if (frequency == null)
53         {
54             return false;
55         }
56         return !_equalityComparer.Equals(frequency.Frequency, default);
57     }
58 }
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             {

```



```

29         Increment(Links.GetOrCreate(sequence[i - 1], sequence[i]));
30     }
31     return indexed;
32 }
33
34 private bool IsIndexedWithIncrement(TLink source, TLink target)
35 {
36     var link = Links.SearchOrCreate(source, target);
37     var indexed = !_equalityComparer.Equals(link, default);
38     if (indexed)
39     {
40         Increment(link);
41     }
42     return indexed;
43 }
44
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.SearchOrCreate(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.SearchOrCreate(sequence[i - 1], sequence[i]),
34                 ↪ default))) { }
35         }
36     }
37 }

```

```

30         return indexed;
31     }
32 }
33 }

```

./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/Indexes/Unindex.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 Unindex<TLink> : ISequenceIndex<TLink>
8      {
9          public virtual bool Add(IList<TLink> sequence) => false;
10
11          public virtual bool MightContain(IList<TLink> sequence) => true;
12      }
13 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;

```

```

3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences
7 {
8     public class ListFiller<TElement, TReturnConstant>
9     {
10         protected readonly List<TElement> _list;
11         protected readonly TReturnConstant _returnConstant;
12
13         public ListFiller(List<TElement> list, TReturnConstant returnConstant)
14         {
15             _list = list;
16             _returnConstant = returnConstant;
17         }
18
19         public ListFiller(List<TElement> list) : this(list, default) { }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public void Add(TElement element) => _list.Add(element);
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public bool AddAndReturnTrue(TElement element)
26         {
27             _list.Add(element);
28             return true;
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public bool AddFirstAndReturnTrue(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  /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
29  /// → графа)
30  ///
31  /// х*у - найти все связи между, в последовательностях любой формы, если не стоит
32  /// → ограничитель на то, что является последовательностью, а что нет,
33  /// то находятся любые структуры связей, которые содержат эти элементы именно в таком
34  /// → порядке.
35  ///
36  /// Рост последовательности слева и справа.
37  /// Поиск со звёздочкой.
38  /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
39  /// так же проблема может быть решена при реализации дистанционных триггеров.
40  /// Нужны ли уникальные указатели вообще?
41  /// Что если обращение к информации будет происходить через содержимое всегда?
42  ///
43  /// Писать тесты.
44  ///
45  /// Можно убрать зависимость от конкретной реализации Links,
46  /// на зависимость от абстрактного элемента, который может быть представлен несколькими
47  /// → способами.
48  ///
49  /// Можно ли как-то сделать один общий интерфейс
50  ///
51  /// Блокчейн и/или гит для распределённой записи транзакций.
52  /// </remarks>
53  public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
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)

```

```

92     {
93         if (Options.UseSequenceMarker)
94         {
95             return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
96         }
97         return sequence;
98     }
99
100 private LinkIndex GetSequenceElements(LinkIndex sequence)
101 {
102     if (Options.UseSequenceMarker)
103     {
104         var linkContents = new UInt64Link(Links.GetLink(sequence));
105         if (linkContents.Source == Options.SequenceMarkerLink)
106         {
107             return linkContents.Target;
108         }
109         if (linkContents.Target == Options.SequenceMarkerLink)
110         {
111             return linkContents.Source;
112         }
113     }
114     return sequence;
115 }
116
117 #region Count
118
119 public LinkIndex Count(IList<LinkIndex> restrictions)
120 {
121     if (restrictions.IsNullOrEmpty())
122     {
123         return Links.Count(Constants.Any, Options.SequenceMarkerLink, Constants.Any);
124     }
125     if (restrictions.Count == 1) // Первая связь это адрес
126     {
127         var sequenceIndex = restrictions[0];
128         if (sequenceIndex == Constants.Null)
129         {
130             return 0;
131         }
132         if (sequenceIndex == Constants.Any)
133         {
134             return Count(null);
135         }
136         if (Options.UseSequenceMarker)
137         {
138             return Links.Count(Constants.Any, Options.SequenceMarkerLink, sequenceIndex);
139         }
140         return Links.Exists(sequenceIndex) ? 1UL : 0;
141     }
142     throw new NotImplementedException();
143 }
144
145 private LinkIndex CountUsages(params LinkIndex[] restrictions)
146 {
147     if (restrictions.Length == 0)
148     {
149         return 0;
150     }
151     if (restrictions.Length == 1) // Первая связь это адрес
152     {
153         if (restrictions[0] == Constants.Null)
154         {
155             return 0;
156         }
157         if (Options.UseSequenceMarker)
158         {
159             var elementsLink = GetSequenceElements(restrictions[0]);
160             var sequenceLink = GetSequenceByElements(elementsLink);
161             if (sequenceLink != Constants.Null)
162             {
163                 return Links.Count(sequenceLink) + Links.Count(elementsLink) - 1;
164             }
165             return Links.Count(elementsLink);
166         }
167         return Links.Count(restrictions[0]);
168     }
169     throw new NotImplementedException();
170 }

```

```

171 #endregion
172
173 #region Create
174
175 public LinkIndex Create(IList<LinkIndex> restrictions)
176 {
177     return _sync.ExecuteWriteOperation(() =>
178     {
179         if (restrictions.IsNullOrEmpty())
180         {
181             return Constants.Null;
182         }
183         Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
184         return CreateCore(restrictions);
185     });
186 }
187
188 private LinkIndex CreateCore(IList<LinkIndex> restrictions)
189 {
190     LinkIndex[] sequence = restrictions.ExtractValues();
191     if (Options.UseIndex)
192     {
193         Options.Index.Add(sequence);
194     }
195     var sequenceRoot = default(LinkIndex);
196     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
197     {
198         var matches = Each(restrictions);
199         if (matches.Count > 0)
200         {
201             sequenceRoot = matches[0];
202         }
203     }
204     else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
205     {
206         return CompactCore(sequence);
207     }
208     if (sequenceRoot == default)
209     {
210         sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
211     }
212     if (Options.UseSequenceMarker)
213     {
214         Links.Unsync.CreateAndUpdate(Options.SequenceMarkerLink, sequenceRoot);
215     }
216     return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
217 }
218
219 #endregion
220
221 #region Each
222
223 public List<LinkIndex> Each(IList<LinkIndex> sequence)
224 {
225     var results = new List<LinkIndex>();
226     var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
227     Each(filler.AddFirstAndReturnConstant, sequence);
228     return results;
229 }
230
231 public LinkIndex Each(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
232     ↪ 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,
250             ↪ Options.SequenceMarkerLink, any));
251     }
252     else
253     {
254         return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
255             ↪ any));
256     }
257     }
258     var sequence =
259     ↪ Options.Walker.Walk(link).ToArray().ConvertToRestrictionsValues();
260     sequence[0] = link;
261     return handler(sequence);
262 }
263 else if (restrictions.Count == 2)
264 {
265     throw new NotImplementedException();
266 }
267 else if (restrictions.Count == 3)
268 {
269     return Links.Unsync.Each(handler, restrictions);
270 }
271 else
272 {
273     var sequence = restrictions.ExtractValues();
274     if (Options.UseIndex && !Options.Index.MightContain(sequence))
275     {
276         return Constants.Break;
277     }
278     return EachCore(handler, sequence);
279 }
280 });
281 }
282
283 private LinkIndex EachCore(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
284     ↪ values)
285 {
286     var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
287     // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
288     ↪ Id.
289     Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
290     ↪ (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
291     ↪ matcher.HandleFullMatched;
292     //if (sequence.Length >= 2)
293     if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
294     {
295         return Constants.Break;
296     }
297     var last = values.Count - 2;
298     for (var i = 1; i < last; i++)
299     {
300         if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
301             ↪ Constants.Continue)
302         {
303             return Constants.Break;
304         }
305     }
306     if (values.Count >= 3)
307     {
308         if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
309             ↪ != Constants.Continue)
310         {
311             return Constants.Break;
312         }
313     }
314     return Constants.Continue;
315 }
316
317 private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
318     ↪ left, LinkIndex right)
319 {
320     return Links.Unsync.Each(doublet =>
321     {
322         var doubletIndex = doublet[Constants.IndexPart];
323         if (StepRight(handler, doubletIndex, right) != Constants.Continue)
324         {
325             return Constants.Break;
326         }
327     }
328 )

```

```

317         if (left != doubletIndex)
318         {
319             return PartialStepRight(handler, doubletIndex, right);
320         }
321         return Constants.Continue;
322     }, new Link<LinkIndex>(Constants.Any, Constants.Any, left));
323 }
324
325 private LinkIndex StepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
    ↳ LinkIndex right) => Links.Unsync.Each(rightStep => TryStepRightUp(handler, right,
    ↳ rightStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, left,
    ↳ Constants.Any));
326
327 private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↳ right, LinkIndex stepFrom)
328 {
329     var upStep = stepFrom;
330     var firstSource = Links.Unsync.GetTarget(upStep);
331     while (firstSource != right && firstSource != upStep)
332     {
333         upStep = firstSource;
334         firstSource = Links.Unsync.GetSource(upStep);
335     }
336     if (firstSource == right)
337     {
338         return handler(new LinkAddress<LinkIndex>(stepFrom));
339     }
340     return Constants.Continue;
341 }
342
343 private LinkIndex StepLeft(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
    ↳ LinkIndex right) => Links.Unsync.Each(leftStep => TryStepLeftUp(handler, left,
    ↳ leftStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, Constants.Any,
    ↳ right));
344
345 private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↳ left, LinkIndex stepFrom)
346 {
347     var upStep = stepFrom;
348     var firstTarget = Links.Unsync.GetSource(upStep);
349     while (firstTarget != left && firstTarget != upStep)
350     {
351         upStep = firstTarget;
352         firstTarget = Links.Unsync.GetTarget(upStep);
353     }
354     if (firstTarget == left)
355     {
356         return handler(new LinkAddress<LinkIndex>(stepFrom));
357     }
358     return Constants.Continue;
359 }
360
361 #endregion
362
363 #region Update
364
365 public LinkIndex Update(IList<LinkIndex> restrictions, IList<LinkIndex> substitution)
366 {
367     var sequence = restrictions.ExtractValues();
368     var newSequence = substitution.ExtractValues();
369
370     if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
371     {
372         return Constants.Null;
373     }
374     if (sequence.IsNullOrEmpty())
375     {
376         return Create(substitution);
377     }
378     if (newSequence.IsNullOrEmpty())
379     {
380         Delete(restrictions);
381         return Constants.Null;
382     }
383     return _sync.ExecuteWriteOperation(() =>
384     {
385         Links.EnsureEachLinkIsAnyOrExists(sequence);
386         Links.EnsureEachLinkExists(newSequence);
387         return UpdateCore(sequence, newSequence);

```



```

388     });
389 }
390
391 private LinkIndex UpdateCore(LinkIndex[] sequence, LinkIndex[] newSequence)
392 {
393     LinkIndex bestVariant;
394     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
395         ↪ !sequence.EqualTo(newSequence))
396     {
397         bestVariant = CompactCore(newSequence);
398     }
399     else
400     {
401         bestVariant = CreateCore(newSequence);
402     }
403     // TODO: Check all options only ones before loop execution
404     // Возможно нужно две версии Each, возвращающий фактические последовательности и с
405     ↪ маркером,
406     // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
407     ↪ можно получить имея только фактические последовательности.
408     foreach (var variant in Each(sequence))
409     {
410         if (variant != bestVariant)
411         {
412             UpdateOneCore(variant, bestVariant);
413         }
414     }
415     return bestVariant;
416 }
417
418 private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
419 {
420     if (Options.UseGarbageCollection)
421     {
422         var sequenceElements = GetSequenceElements(sequence);
423         var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
424         var sequenceLink = GetSequenceByElements(sequenceElements);
425         var newSequenceElements = GetSequenceElements(newSequence);
426         var newSequenceLink = GetSequenceByElements(newSequenceElements);
427         if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
428         {
429             if (sequenceLink != Constants.Null)
430             {
431                 Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
432             }
433             Links.Unsync.MergeUsages(sequenceElements, newSequenceElements);
434         }
435         ClearGarbage(sequenceElementsContents.Source);
436         ClearGarbage(sequenceElementsContents.Target);
437     }
438     else
439     {
440         if (Options.UseSequenceMarker)
441         {
442             var sequenceElements = GetSequenceElements(sequence);
443             var sequenceLink = GetSequenceByElements(sequenceElements);
444             var newSequenceElements = GetSequenceElements(newSequence);
445             var newSequenceLink = GetSequenceByElements(newSequenceElements);
446             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
447             {
448                 if (sequenceLink != Constants.Null)
449                 {
450                     Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
451                 }
452                 Links.Unsync.MergeUsages(sequenceElements, newSequenceElements);
453             }
454         }
455         else
456         {
457             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
458             {
459                 Links.Unsync.MergeUsages(sequence, newSequence);
460             }
461         }
462     }
463 }
464
465 #endregion

```

```

463 #region Delete
464
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,
    ↪ sequence);
548
549 #endregion
550
551 #region Garbage Collection
552
553 /// <remarks>
554 /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
    ↪ определить извне или в унаследованном классе
555 /// </remarks>
556 [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 {
615     _filterPosition = 0;
616     foreach (var part in Walk(sequenceToMatch))
617     {
618         if (!FullMatchCore(part))
619         {
620             break;
621         }
622     }
623     return _filterPosition == _patternSequence.Count;
624 }
625
626 private bool FullMatchCore(LinkIndex element)
627 {
628     if (_filterPosition == _patternSequence.Count)
629     {
630         _filterPosition = -2; // Длиннее чем нужно
631         return false;
632     }
633     if (_patternSequence[_filterPosition] != Links.Constants.Any
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))

```

```

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

```

```

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

```

```

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

```



```

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

```

```

313 private void PartialStepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
314 {
315     Links.Unsync.Each(right, Constants.Any, doublet =>
316     {
317         StepLeft(handler, left, doublet);
318         if (right != doublet)
319         {
320             PartialStepLeft(handler, left, doublet);
321         }
322         return true;
323     });
324 }
325
326 private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
327 {
328     Links.Unsync.Each(Constants.Any, right, leftStep =>
329     {
330         TryStepLeftUp(handler, left, leftStep);
331         return true;
332     });
333 }
334
335 private void TryStepLeftUp(Action<IList<LinkIndex>> handler, ulong left, ulong stepFrom)
336 {
337     var upStep = stepFrom;
338     var firstTarget = Links.Unsync.GetSource(upStep);
339     while (firstTarget != left && firstTarget != upStep)
340     {
341         upStep = firstTarget;
342         firstTarget = Links.Unsync.GetTarget(upStep);
343     }
344     if (firstTarget == left)
345     {
346         handler(new LinkAddress<LinkIndex>(stepFrom));
347     }
348 }
349
350 private bool StartsWith(ulong sequence, ulong link)
351 {
352     var upStep = sequence;
353     var firstSource = Links.Unsync.GetSource(upStep);
354     while (firstSource != link && firstSource != upStep)
355     {
356         upStep = firstSource;
357         firstSource = Links.Unsync.GetSource(upStep);
358     }
359     return firstSource == link;
360 }
361
362 private bool EndsWith(ulong sequence, ulong link)
363 {
364     var upStep = sequence;
365     var lastTarget = Links.Unsync.GetTarget(upStep);
366     while (lastTarget != link && lastTarget != upStep)
367     {
368         upStep = lastTarget;
369         lastTarget = Links.Unsync.GetTarget(upStep);
370     }
371     return lastTarget == link;
372 }
373
374 public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
375 {
376     return _sync.ExecuteReadOperation(() =>
377     {
378         var results = new List<ulong>();
379         if (sequence.Length > 0)
380         {
381             Links.EnsureEachLinkExists(sequence);
382             var firstElement = sequence[0];
383             if (sequence.Length == 1)
384             {
385                 results.Add(firstElement);
386                 return results;
387             }
388             if (sequence.Length == 2)
389             {
390                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
391                 if (doublet != Constants.Null)

```

```

392         {
393             results.Add(doublet);
394         }
395         return results;
396     }
397     var linksInSequence = new HashSet<ulong>(sequence);
398     void handler(ICollection<LinkIndex> result)
399     {
400         var resultIndex = result[Links.Constants.IndexPart];
401         var filterPosition = 0;
402         StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
403             ↪ Links.Unsync.GetTarget,
404             x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
405             ↪ x =>
406             {
407                 if (filterPosition == sequence.Length)
408                 {
409                     filterPosition = -2; // Длиннее чем нужно
410                     return false;
411                 }
412                 if (x != sequence[filterPosition])
413                 {
414                     filterPosition = -1;
415                     return false; // Начинается иначе
416                 }
417                 filterPosition++;
418                 return true;
419             });
420         if (filterPosition == sequence.Length)
421         {
422             results.Add(resultIndex);
423         }
424     }
425     if (sequence.Length >= 2)
426     {
427         StepRight(handler, sequence[0], sequence[1]);
428     }
429     var last = sequence.Length - 2;
430     for (var i = 1; i < last; i++)
431     {
432         PartialStepRight(handler, sequence[i], sequence[i + 1]);
433     }
434     if (sequence.Length >= 3)
435     {
436         StepLeft(handler, sequence[sequence.Length - 2],
437             ↪ sequence[sequence.Length - 1]);
438     }
439     }
440     return results;
441 });
442 }
443
444 public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
445 {
446     return _sync.ExecuteReadOperation(() =>
447     {
448         var results = new HashSet<ulong>();
449         if (sequence.Length > 0)
450         {
451             Links.EnsureEachLinkExists(sequence);
452             var firstElement = sequence[0];
453             if (sequence.Length == 1)
454             {
455                 results.Add(firstElement);
456                 return results;
457             }
458             if (sequence.Length == 2)
459             {
460                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
461                 if (doublet != Constants.Null)
462                 {
463                     results.Add(doublet);
464                 }
465                 return results;
466             }
467             var matcher = new Matcher(this, sequence, results, null);
468             if (sequence.Length >= 2)
469             {

```

```

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

```

```

530 {
531     var linksInSequence = new HashSet<ulong>(knownElements);
532     var entered = new HashSet<ulong>();
533     var sb = new StringBuilder();
534     sb.Append('{');
535     if (links.Exists(sequenceLink))
536     {
537         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
538             x => linksInSequence.Contains(x) || links.IsFullPoint(x),
539             ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
540             {
541                 if (insertComma && sb.Length > 1)
542                 {
543                     sb.Append(',');
544                 }
545                 if (entered.Contains(element))
546                 {
547                     sb.Append('{');
548                     elementToString(sb, element);
549                     sb.Append('}');
550                 }
551                 else
552                 {
553                     elementToString(sb, element);
554                 }
555                 if (sb.Length < MaxSequenceFormatSize)
556                 {
557                     return true;
558                 }
559                 sb.Append(insertComma ? ", ..." : "...");
560                 return false;
561             });
562     }
563     sb.Append('}');
564     return sb.ToString();
565 }
566
567 public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
568 {
569     return _sync.ExecuteReadOperation(() =>
570     {
571         if (sequence.Length > 0)
572         {
573             Links.EnsureEachLinkExists(sequence);
574             var results = new HashSet<ulong>();
575             for (var i = 0; i < sequence.Length; i++)
576             {
577                 AllUsagesCore(sequence[i], results);
578             }
579             var filteredResults = new List<ulong>();
580             var linksInSequence = new HashSet<ulong>(sequence);
581             foreach (var result in results)
582             {
583                 var filterPosition = -1;
584                 StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
585                     ↪ Links.Unsync.GetTarget,
586                     x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
587                     ↪ x =>
588                     {
589                         if (filterPosition == (sequence.Length - 1))
590                         {
591                             return false;
592                         }
593                         if (filterPosition >= 0)
594                         {
595                             if (x == sequence[filterPosition + 1])
596                             {
597                                 filterPosition++;
598                             }
599                             else
600                             {
601                                 return false;
602                             }
603                         }
604                     }
605                 }
606                 if (filterPosition < 0)
607                 {
608                     if (x == sequence[0])
609                     {
610

```

```

605         filterPosition = 0;
606     }
607 }
608     return true;
609 });
610     if (filterPosition == (sequence.Length - 1))
611     {
612         filteredResults.Add(result);
613     }
614 }
615     return filteredResults;
616 }
617     return new List<ulong>();
618 });
619 }
620
621 public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
622 {
623     return _sync.ExecuteReadOperation(() =>
624     {
625         if (sequence.Length > 0)
626         {
627             Links.EnsureEachLinkExists(sequence);
628             var results = new HashSet<ulong>();
629             for (var i = 0; i < sequence.Length; i++)
630             {
631                 AllUsagesCore(sequence[i], results);
632             }
633             var filteredResults = new HashSet<ulong>();
634             var matcher = new Matcher(this, sequence, filteredResults, null);
635             matcher.AddAllPartialMatchedToResults(results);
636             return filteredResults;
637         }
638         return new HashSet<ulong>();
639     });
640 }
641
642 public bool GetAllPartiallyMatchingSequences2(Func<IList<LinkIndex>, LinkIndex> handler,
643     → params ulong[] sequence)
644 {
645     return _sync.ExecuteReadOperation(() =>
646     {
647         if (sequence.Length > 0)
648         {
649             Links.EnsureEachLinkExists(sequence);
650
651             var results = new HashSet<ulong>();
652             var filteredResults = new HashSet<ulong>();
653             var matcher = new Matcher(this, sequence, filteredResults, handler);
654             for (var i = 0; i < sequence.Length; i++)
655             {
656                 if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
657                 {
658                     return false;
659                 }
660             }
661             return true;
662         }
663         return true;
664     });
665 }
666
667 //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
668 //{
669 //    return Sync.ExecuteReadOperation(() =>
670 //    {
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         //             firstResults.IntersectWith(lastResults);
684
685         //             //for (var i = 0; i < sequence.Length; i++)
686         //             //     AllUsagesCore(sequence[i], results);
687
688         //             var filteredResults = new HashSet<ulong>();
689         //             var matcher = new Matcher(this, sequence, filteredResults, null);
690         //             matcher.AddAllPartialMatchedToResults(firstResults);
691         //             return filteredResults;
692         //         }
693
694         //     return new HashSet<ulong>();
695     });
696 //}
697
698 public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
699 {
700     return _sync.ExecuteReadOperation(() =>
701     {
702         if (sequence.Length > 0)
703         {
704             Links.EnsureEachLinkIsAnyOrExists(sequence);
705             var firstResults = new HashSet<ulong>();
706             var lastResults = new HashSet<ulong>();
707             var first = sequence.First(x => x != Constants.Any);
708             var last = sequence.Last(x => x != Constants.Any);
709             AllUsagesCore(first, firstResults);
710             AllUsagesCore(last, lastResults);
711             firstResults.IntersectWith(lastResults);
712             //for (var i = 0; i < sequence.Length; i++)
713             //     AllUsagesCore(sequence[i], results);
714             var filteredResults = new HashSet<ulong>();
715             var matcher = new Matcher(this, sequence, filteredResults, null);
716             matcher.AddAllPartialMatchedToResults(firstResults);
717             return filteredResults;
718         }
719         return new HashSet<ulong>();
720     });
721 }
722
723 public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
724 ↪ IList<ulong> sequence)
725 {
726     return _sync.ExecuteReadOperation(() =>
727     {
728         if (sequence.Count > 0)
729         {
730             Links.EnsureEachLinkExists(sequence);
731             var results = new HashSet<LinkIndex>();
732             //var nextResults = new HashSet<ulong>();
733             //for (var i = 0; i < sequence.Length; i++)
734             //{
735             //     AllUsagesCore(sequence[i], nextResults);
736             //     if (results.IsNullOrEmpty())
737             //     {
738             //         results = nextResults;
739             //         nextResults = new HashSet<ulong>();
740             //     }
741             //     else
742             //     {
743             //         results.IntersectWith(nextResults);
744             //         nextResults.Clear();
745             //     }
746             //}
747             var collector1 = new AllUsagesCollector1(Links.Unsync, results);
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);

```

```

759         matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
760             ↪ x)); // OrderBy is a Hack
761         return filteredResults;
762     }
763     return new HashSet<ulong>();
764 }
765
766 // Does not work
767 //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
768     ↪ params ulong[] sequence)
769 //{
770 //    var visited = new HashSet<ulong>();
771 //    var results = new HashSet<ulong>();
772 //    var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
773     ↪ true; }, readAsElements);
774 //    var last = sequence.Length - 1;
775 //    for (var i = 0; i < last; i++)
776 //    {
777 //        PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
778 //    }
779 //    return results;
780 //}
781
782 public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
783 {
784     return _sync.ExecuteReadOperation(() =>
785     {
786         if (sequence.Length > 0)
787         {
788             Links.EnsureEachLinkExists(sequence);
789             //var firstElement = sequence[0];
790             //if (sequence.Length == 1)
791             //{
792             //    //results.Add(firstElement);
793             //    return results;
794             //}
795             //if (sequence.Length == 2)
796             //{
797             //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
798             //    //if (doublet != Doublets.Links.Null)
799             //    //    results.Add(doublet);
800             //    return results;
801             //}
802             //var lastElement = sequence[sequence.Length - 1];
803             //Func<ulong, bool> handler = x =>
804             //{
805             //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
806             //        ↪ results.Add(x);
807             //    return true;
808             //};
809             //if (sequence.Length >= 2)
810             //    StepRight(handler, sequence[0], sequence[1]);
811             //var last = sequence.Length - 2;
812             //for (var i = 1; i < last; i++)
813             //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
814             //if (sequence.Length >= 3)
815             //    StepLeft(handler, sequence[sequence.Length - 2],
816             //        ↪ sequence[sequence.Length - 1]);
817             //if (sequence.Length == 1)
818             //    throw new NotImplementedException(); // all sequences, containing
819             //    ↪ this element?
820             //if (sequence.Length == 2)
821             //{
822             //    var results = new List<ulong>();
823             //    PartialStepRight(results.Add, sequence[0], sequence[1]);
824             //    return results;
825             //}
826             //var matches = new List<List<ulong>>();
827             //var last = sequence.Length - 1;
828             //for (var i = 0; i < last; i++)
829             //{
830             //    var results = new List<ulong>();
831             //    //StepRight(results.Add, sequence[i], sequence[i + 1]);
832             //    PartialStepRight(results.Add, sequence[i], sequence[i + 1]);

```



```

830         if (results.Count > 0)
831             matches.Add(results);
832         else
833             return results;
834         if (matches.Count == 2)
835         {
836             var merged = new List<ulong>();
837             for (var j = 0; j < matches[0].Count; j++)
838                 for (var k = 0; k < matches[1].Count; k++)
839                     CloseInnerConnections(merged.Add, matches[0][j],
840                     ↪ matches[1][k]);
841             if (merged.Count > 0)
842                 matches = new List<List<ulong>> { merged };
843             else
844                 return new List<ulong>();
845         }
846         if (matches.Count > 0)
847         {
848             var usages = new HashSet<ulong>();
849             for (int i = 0; i < sequence.Length; i++)
850             {
851                 AllUsagesCore(sequence[i], usages);
852             }
853             //for (int i = 0; i < matches[0].Count; i++)
854             //    AllUsagesCore(matches[0][i], usages);
855             //usages.UnionWith(matches[0]);
856             return usages.ToList();
857         }
858         var firstLinkUsages = new HashSet<ulong>();
859         AllUsagesCore(sequence[0], firstLinkUsages);
860         firstLinkUsages.Add(sequence[0]);
861         //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
862         ↪ sequence[0] }; // or all sequences, containing this element?
863         //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
864         ↪ 1).ToList();
865         var results = new HashSet<ulong>();
866         foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
867         ↪ firstLinkUsages, 1))
868         {
869             AllUsagesCore(match, results);
870         }
871         return results.ToList();
872     }
873     return new List<ulong>();
874 });
875 }
876
877 /// <remarks>
878 /// TODO: Может потребоваться ограничение на уровень глубины рекурсии
879 /// </remarks>
880 public HashSet<ulong> AllUsages(ulong link)
881 {
882     return _sync.ExecuteReadOperation(() =>
883     {
884         var usages = new HashSet<ulong>();
885         AllUsagesCore(link, usages);
886         return usages;
887     });
888 }
889
890 // При сборе всех использований (последовательностей) можно сохранять обратный путь к
891 ↪ той связи с которой начинался поиск (STTTSSSTT),
892 // причём достаточно одного бита для хранения перехода влево или вправо
893 private void AllUsagesCore(ulong link, HashSet<ulong> usages)
894 {
895     bool handler(ulong doublet)
896     {
897         if (usages.Add(doublet))
898         {
899             AllUsagesCore(doublet, usages);
900         }
901         return true;
902     }
903     Links.Unsync.Each(link, Constants.Any, handler);
904     Links.Unsync.Each(Constants.Any, link, handler);
905 }

```

```

902 public HashSet<ulong> AllBottomUsages(ulong link)
903 {
904     return _sync.ExecuteReadOperation(() =>
905     {
906         var visits = new HashSet<ulong>();
907         var usages = new HashSet<ulong>();
908         AllBottomUsagesCore(link, visits, usages);
909         return usages;
910     });
911 }
912
913 private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
914     ↳ usages)
915 {
916     bool handler(ulong doublet)
917     {
918         if (visits.Add(doublet))
919         {
920             AllBottomUsagesCore(doublet, visits, usages);
921         }
922         return true;
923     }
924     if (Links.Unsync.Count(Constants.Any, link) == 0)
925     {
926         usages.Add(link);
927     }
928     else
929     {
930         Links.Unsync.Each(link, Constants.Any, handler);
931         Links.Unsync.Each(Constants.Any, link, handler);
932     }
933 }
934
935 public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
936 {
937     if (Options.UseSequenceMarker)
938     {
939         var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
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 }

```

```

976 public void CalculateAllUsages2(ulong[] totals)
977 {
978     var calculator = new AllUsagesCalculator2(Links, totals);
979     calculator.Calculate();
980 }
981
982 private class AllUsagesCalculator
983 {
984     private readonly SynchronizedLinks<ulong> _links;
985     private readonly ulong[] _totals;
986
987     public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
988     {
989         _links = links;
990         _totals = totals;
991     }
992
993     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
994         ↪ CalculateCore);
995
996     private bool CalculateCore(ulong link)
997     {
998         if (_totals[link] == 0)
999         {
1000             var total = 1UL;
1001             _totals[link] = total;
1002             var visitedChildren = new HashSet<ulong>();
1003             bool linkCalculator(ulong child)
1004             {
1005                 if (link != child && visitedChildren.Add(child))
1006                 {
1007                     total += _totals[child] == 0 ? 1 : _totals[child];
1008                 }
1009                 return true;
1010             }
1011             _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);
1012             _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
1013             _totals[link] = total;
1014         }
1015         return true;
1016     }
1017 }
1018
1019 private class AllUsagesCalculator2
1020 {
1021     private readonly SynchronizedLinks<ulong> _links;
1022     private readonly ulong[] _totals;
1023
1024     public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1025     {
1026         _links = links;
1027         _totals = totals;
1028     }
1029
1030     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
1031         ↪ CalculateCore);
1032
1033     private bool IsElement(ulong link)
1034     {
1035         // _linksInSequence.Contains(link) ||
1036         return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
1037             ↪ link;
1038     }
1039
1040     private bool CalculateCore(ulong link)
1041     {
1042         // TODO: Проработать защиту от заикливания
1043         // Основано на SequenceWalker.WalkLeft
1044         Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1045         Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1046         Func<ulong, bool> isElement = IsElement;
1047         void visitLeaf(ulong parent)
1048         {
1049             if (link != parent)
1050             {
1051                 _totals[parent]++;
1052             }
1053         }
1054         void visitNode(ulong parent)

```

```

1052     {
1053         if (link != parent)
1054         {
1055             _totals[parent]++;
1056         }
1057     }
1058     var stack = new Stack();
1059     var element = link;
1060     if (isElement(element))
1061     {
1062         visitLeaf(element);
1063     }
1064     else
1065     {
1066         while (true)
1067         {
1068             if (isElement(element))
1069             {
1070                 if (stack.Count == 0)
1071                 {
1072                     break;
1073                 }
1074                 element = stack.Pop();
1075                 var source = getSource(element);
1076                 var target = getTarget(element);
1077                 // Überprüfung 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(IList<ulong> link)
1137     {
1138         var linkIndex = _links.GetIndex(link);
1139         if (_usages.Add(linkIndex))
1140         {
1141             _links.Each(Collect, _links.Constants.Any, linkIndex);
1142         }
1143         return _continue;
1144     }
1145 }
1146
1147 private class AllUsagesCollector2
1148 {
1149     private readonly ILinks<ulong> _links;
1150     private readonly BitString _usages;
1151
1152     public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
1153     {
1154         _links = links;
1155         _usages = usages;
1156     }
1157
1158     public bool Collect(ulong link)
1159     {
1160         if (_usages.Add((long)link))
1161         {
1162             _links.Each(link, _links.Constants.Any, Collect);
1163             _links.Each(_links.Constants.Any, link, Collect);
1164         }
1165         return true;
1166     }
1167 }
1168
1169 private class AllUsagesIntersectingCollector
1170 {
1171     private readonly SynchronizedLinks<ulong> _links;
1172     private readonly HashSet<ulong> _intersectWith;
1173     private readonly HashSet<ulong> _usages;
1174     private readonly HashSet<ulong> _enter;
1175
1176     public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
1177     ↪ intersectWith, HashSet<ulong> usages)
1178     {
1179         _links = links;
1180         _intersectWith = intersectWith;
1181         _usages = usages;
1182         _enter = new HashSet<ulong>(); // защита от зацикливания
1183     }
1184
1185     public bool Collect(ulong link)
1186     {
1187         if (_enter.Add(link))
1188         {
1189             if (_intersectWith.Contains(link))
1190             {
1191                 _usages.Add(link);
1192             }
1193             _links.Unsync.Each(link, _links.Constants.Any, Collect);
1194             _links.Unsync.Each(_links.Constants.Any, link, Collect);
1195         }
1196         return true;
1197     }
1198 }
1199
1200 private void CloseInnerConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
1201 ↪ right)
1202 {
1203     TryStepLeftUp(handler, left, right);
1204     TryStepRightUp(handler, right, left);
1205 }
1206
1207 private void AllCloseConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
1208 ↪ right)
1209 {
1210     // Direct

```

```

1208     if (left == right)
1209     {
1210         handler(new LinkAddress<LinkIndex>(left));
1211     }
1212     var doublet = Links.Unsync.SearchOrDefault(left, right);
1213     if (doublet != Constants.Null)
1214     {
1215         handler(new LinkAddress<LinkIndex>(doublet));
1216     }
1217     // Inner
1218     CloseInnerConnections(handler, left, right);
1219     // Outer
1220     StepLeft(handler, left, right);
1221     StepRight(handler, left, right);
1222     PartialStepRight(handler, left, right);
1223     PartialStepLeft(handler, left, right);
1224 }
1225
1226 private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
1227     ↪ HashSet<ulong> previousMatchings, long startAt)
1228 {
1229     if (startAt >= sequence.Length) // ?
1230     {
1231         return previousMatchings;
1232     }
1233     var secondLinkUsages = new HashSet<ulong>();
1234     AllUsagesCore(sequence[startAt], secondLinkUsages);
1235     secondLinkUsages.Add(sequence[startAt]);
1236     var matchings = new HashSet<ulong>();
1237     var filler = new SetFiller<LinkIndex, LinkIndex>(matchings, Constants.Continue);
1238     //for (var i = 0; i < previousMatchings.Count; i++)
1239     foreach (var secondLinkUsage in secondLinkUsages)
1240     {
1241         foreach (var previousMatching in previousMatchings)
1242         {
1243             //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
1244             ↪ secondLinkUsage);
1245             StepRight(filler.AddFirstAndReturnConstant, previousMatching,
1246             ↪ secondLinkUsage);
1247             TryStepRightUp(filler.AddFirstAndReturnConstant, secondLinkUsage,
1248             ↪ previousMatching);
1249             //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
1250             ↪ sequence[startAt]); // почему-то эта ошибочная запись приводит к
1251             ↪ желаемым результатам.
1252             PartialStepRight(filler.AddFirstAndReturnConstant, previousMatching,
1253             ↪ secondLinkUsage);
1254         }
1255     }
1256     if (matchings.Count == 0)
1257     {
1258         return matchings;
1259     }
1260     return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
1261 }
1262
1263 private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
1264     ↪ links, params ulong[] sequence)
1265 {
1266     if (sequence == null)
1267     {
1268         return;
1269     }
1270     for (var i = 0; i < sequence.Length; i++)
1271     {
1272         if (sequence[i] != links.Constants.Any && sequence[i] != ZeroOrMany &&
1273             ↪ !links.Exists(sequence[i]))
1274         {
1275             throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
1276             ↪ $"patternSequence[{i}]");
1277         }
1278     }
1279 }
1280
1281 // Pattern Matching -> Key To Triggers
1282 public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1283 {
1284     return _sync.ExecuteReadOperation(() =>

```

```

1275     {
1276         patternSequence = Simplify(patternSequence);
1277         if (patternSequence.Length > 0)
1278         {
1279             EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1280             var uniqueSequenceElements = new HashSet<ulong>();
1281             for (var i = 0; i < patternSequence.Length; i++)
1282             {
1283                 if (patternSequence[i] != Constants.Any && patternSequence[i] !=
1284                     ↪ ZeroOrMany)
1285                 {
1286                     uniqueSequenceElements.Add(patternSequence[i]);
1287                 }
1288             }
1289             var results = new HashSet<ulong>();
1290             foreach (var uniqueSequenceElement in uniqueSequenceElements)
1291             {
1292                 AllUsagesCore(uniqueSequenceElement, results);
1293             }
1294             var filteredResults = new HashSet<ulong>();
1295             var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1296             matcher.AddAllPatternMatchedToResults(results);
1297             return filteredResults;
1298         }
1299         return new HashSet<ulong>();
1300     });
1301 }
1302
1303 // Найти все возможные связи между указанным списком связей.
1304 // Находит связи между всеми указанными связями в любом порядке.
1305 // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
1306 ↪ несколько раз в последовательности)
1307 public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1308 {
1309     return _sync.ExecuteReadOperation(() =>
1310     {
1311         var results = new HashSet<ulong>();
1312         if (linksToConnect.Length > 0)
1313         {
1314             Links.EnsureEachLinkExists(linksToConnect);
1315             AllUsagesCore(linksToConnect[0], results);
1316             for (var i = 1; i < linksToConnect.Length; i++)
1317             {
1318                 var next = new HashSet<ulong>();
1319                 AllUsagesCore(linksToConnect[i], next);
1320                 results.IntersectWith(next);
1321             }
1322             return results;
1323         }
1324     });
1325 }
1326
1327 public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
1328 {
1329     return _sync.ExecuteReadOperation(() =>
1330     {
1331         var results = new HashSet<ulong>();
1332         if (linksToConnect.Length > 0)
1333         {
1334             Links.EnsureEachLinkExists(linksToConnect);
1335             var collector1 = new AllUsagesCollector(Links.Unsync, results);
1336             collector1.Collect(linksToConnect[0]);
1337             var next = new HashSet<ulong>();
1338             for (var i = 1; i < linksToConnect.Length; i++)
1339             {
1340                 var collector = new AllUsagesCollector(Links.Unsync, next);
1341                 collector.Collect(linksToConnect[i]);
1342                 results.IntersectWith(next);
1343                 next.Clear();
1344             }
1345             return results;
1346         }
1347     });
1348 }
1349
1350 public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
1351 {
1352     return _sync.ExecuteReadOperation(() =>

```

```

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

```



```

1427 //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1428 //if (zeroOrManyStepped && newZeroOrManyStepped)
1429 //    continue;
1430 //zeroOrManyStepped = newZeroOrManyStepped;
1431 if (sequence[i] == ZeroOrMany)
1432 {
1433     if (zeroOrManyStepped)
1434     {
1435         continue;
1436     }
1437     zeroOrManyStepped = true;
1438 }
1439 else
1440 {
1441     //if (zeroOrManyStepped) Is it efficient?
1442     zeroOrManyStepped = false;
1443 }
1444 newSequence[j++] = sequence[i];
1445 }
1446 return newSequence;
1447 }
1448
1449 public static void TestSimplify()
1450 {
1451     var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany,
1452     ↪ ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
1453     var simplifiedSequence = Simplify(sequence);
1454 }
1455
1456 public List<ulong> GetSimilarSequences() => new List<ulong>();
1457
1458 public void Prediction()
1459 {
1460     //_links
1461     //_sequences
1462 }
1463
1464 #region From Triplets
1465
1466 //public static void DeleteSequence(Link sequence)
1467 //{
1468 //}
1469
1470 public List<ulong> CollectMatchingSequences(ulong[] links)
1471 {
1472     if (links.Length == 1)
1473     {
1474         throw new Exception("Подпоследовательности с одним элементом не
1475         ↪ поддерживаются.");
1476     }
1477     var leftBound = 0;
1478     var rightBound = links.Length - 1;
1479     var left = links[leftBound++];
1480     var right = links[rightBound--];
1481     var results = new List<ulong>();
1482     CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
1483     return results;
1484 }
1485
1486 private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
1487     ↪ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1488 {
1489     var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1490     var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1491     if (leftLinkTotalReferers <= rightLinkTotalReferers)
1492     {
1493         var nextLeftLink = middleLinks[leftBound];
1494         var elements = GetRightElements(leftLink, nextLeftLink);
1495         if (leftBound <= rightBound)
1496         {
1497             for (var i = elements.Length - 1; i >= 0; i--)
1498             {
1499                 var element = elements[i];
1500                 if (element != 0)
1501                 {
1502                     CollectMatchingSequences(element, leftBound + 1, middleLinks,
1503                     ↪ rightLink, rightBound, ref results);
1504                 }
1505             }
1506         }
1507     }
1508 }

```

```

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
    ↪ == Net.And &&
1583 {
1584     result[offset + 1] = couple;
1585     if (++added == 2)
1586     {
1587         return false;
1588     }
1589 }
1590 }
1591 return true;
1592 });
1593 return added > 0;
1594 }
1595
1596 public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
1597 {
1598     var result = new ulong[5];
1599     TryStepLeft(startLink, leftLink, result, 0);
1600     Links.Each(startLink, Constants.Any, couple =>
1601     {
1602         if (couple != startLink)
1603         {
1604             if (TryStepLeft(couple, leftLink, result, 2))
1605             {
1606                 return false;
1607             }
1608         }
1609         return true;
1610     });
1611     if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
1612     {
1613         result[4] = leftLink;
1614     }
1615     return result;
1616 }
1617
1618 public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
1619 {
1620     var added = 0;
1621     Links.Each(Constants.Any, startLink, couple =>
1622     {
1623         if (couple != startLink)
1624         {
1625             var coupleSource = Links.GetSource(couple);
1626             if (coupleSource == leftLink)
1627             {
1628                 result[offset] = couple;
1629                 if (++added == 2)
1630                 {
1631                     return false;
1632                 }
1633             }
1634             else if (Links.GetTarget(coupleSource) == leftLink) // coupleSource.Linker
1635                 ↪ == Net.And &&
1636             {
1637                 result[offset + 1] = couple;
1638                 if (++added == 2)
1639                 {
1640                     return false;
1641                 }
1642             }
1643         }
1644         return true;
1645     });
1646     return added > 0;
1647 }
1648
1649 #endregion
1650
1651 #region Walkers
1652 public class PatternMatcher : RightSequenceWalker<ulong>
1653 {
1654     private readonly Sequences _sequences;
1655     private readonly ulong[] _patternSequence;

```

```

1656 private readonly HashSet<LinkIndex> _linksInSequence;
1657 private readonly HashSet<LinkIndex> _results;
1658
1659 #region Pattern Match
1660
1661 enum PatternBlockType
1662 {
1663     Undefined,
1664     Gap,
1665     Elements
1666 }
1667
1668 struct PatternBlock
1669 {
1670     public PatternBlockType Type;
1671     public long Start;
1672     public long Stop;
1673 }
1674
1675 private readonly List<PatternBlock> _pattern;
1676 private int _patternPosition;
1677 private long _sequencePosition;
1678
1679 #endregion
1680
1681 public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
1682     ↳ HashSet<LinkIndex> results)
1683     : base(sequences.Links.Unsync, new DefaultStack<ulong>())
1684 {
1685     _sequences = sequences;
1686     _patternSequence = patternSequence;
1687     _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
1688     ↳ _sequences.Constants.Any && x != ZeroOrMany));
1689     _results = results;
1690     _pattern = CreateDetailedPattern();
1691 }
1692
1693 protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
1694     ↳ base.IsElement(link);
1695
1696 public bool PatternMatch(LinkIndex sequenceToMatch)
1697 {
1698     _patternPosition = 0;
1699     _sequencePosition = 0;
1700     foreach (var part in Walk(sequenceToMatch))
1701     {
1702         if (!PatternMatchCore(part))
1703         {
1704             break;
1705         }
1706     }
1707     return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
1708     ↳ - 1 && _pattern[_patternPosition].Start == 0);
1709 }
1710
1711 private List<PatternBlock> CreateDetailedPattern()
1712 {
1713     var pattern = new List<PatternBlock>();
1714     var patternBlock = new PatternBlock();
1715     for (var i = 0; i < _patternSequence.Length; i++)
1716     {
1717         if (patternBlock.Type == PatternBlockType.Undefined)
1718         {
1719             if (_patternSequence[i] == _sequences.Constants.Any)
1720             {
1721                 patternBlock.Type = PatternBlockType.Gap;
1722                 patternBlock.Start = 1;
1723                 patternBlock.Stop = 1;
1724             }
1725             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     }
1739 }

```

```

1733     }
1734     else if (patternBlock.Type == PatternBlockType.Elements)
1735     {
1736         if (_patternSequence[i] == _sequences.Constants.Any)
1737         {
1738             pattern.Add(patternBlock);
1739             patternBlock = new PatternBlock
1740             {
1741                 Type = PatternBlockType.Gap,
1742                 Start = 1,
1743                 Stop = 1
1744             };
1745         }
1746         else if (_patternSequence[i] == ZeroOrMany)
1747         {
1748             pattern.Add(patternBlock);
1749             patternBlock = new PatternBlock
1750             {
1751                 Type = PatternBlockType.Gap,
1752                 Start = 0,
1753                 Stop = long.MaxValue
1754             };
1755         }
1756         else
1757         {
1758             patternBlock.Stop = i;
1759         }
1760     }
1761     else // patternBlock.Type == PatternBlockType.Gap
1762     {
1763         if (_patternSequence[i] == _sequences.Constants.Any)
1764         {
1765             patternBlock.Start++;
1766             if (patternBlock.Stop < patternBlock.Start)
1767             {
1768                 patternBlock.Stop = patternBlock.Start;
1769             }
1770         }
1771         else if (_patternSequence[i] == ZeroOrMany)
1772         {
1773             patternBlock.Stop = long.MaxValue;
1774         }
1775         else
1776         {
1777             pattern.Add(patternBlock);
1778             patternBlock = new PatternBlock
1779             {
1780                 Type = PatternBlockType.Elements,
1781                 Start = i,
1782                 Stop = i
1783             };
1784         }
1785     }
1786 }
1787 if (patternBlock.Type != PatternBlockType.Undefined)
1788 {
1789     pattern.Add(patternBlock);
1790 }
1791 return pattern;
1792 }
1793
1794 // match: search for regexp anywhere in text
1795 //int match(char* regexp, char* text)
1796 //{
1797 //    do
1798 //    {
1799 //        } while (*text++ != '\0');
1800 //    return 0;
1801 //}
1802
1803 // matchhere: search for regexp at beginning of text
1804 //int matchhere(char* regexp, char* text)
1805 //{
1806 //    if (regexp[0] == '\0')
1807 //        return 1;
1808 //    if (regexp[1] == '*')
1809 //        return matchstar(regexp[0], regexp + 2, text);
1810 //    if (regexp[0] == '$' && regexp[1] == '\0')
1811 //        return *text == '\0';

```

```

1812 // if (*text != '\0' && (regex[0] == '.' || regex[0] == *text))
1813 //     return matchhere(regex + 1, text + 1);
1814 // return 0;
1815 //}
1816
1817 // matchstar: search for c*regex at beginning of text
1818 //int matchstar(int c, char* regex, char* text)
1819 //{
1820 //    do
1821 //    { /* a * matches zero or more instances */
1822 //        if (matchhere(regex, text))
1823 //            return 1;
1824 //    } while (*text != '\0' && (*text++ == c || c == '.'));
1825 //    return 0;
1826 //}
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 }

```

```

1889     }
1890 }
1891 }
1892 else // currentPatternBlock.Type == PatternBlockType.Elements
1893 {
1894     var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1895     if (_patternSequence[patternElementPosition] != element)
1896     {
1897         return false; // Соответствие невозможно
1898     }
1899     if (patternElementPosition == currentPatternBlock.Stop)
1900     {
1901         _patternPosition++;
1902         _sequencePosition = 0;
1903     }
1904     else
1905     {
1906         _sequencePosition++;
1907     }
1908 }
1909 return true;
1910 //if (_patternSequence[_patternPosition] != element)
1911 //    return false;
1912 //else
1913 //{
1914 //    _sequencePosition++;
1915 //    _patternPosition++;
1916 //    return true;
1917 //}
1918 ///////
1919 //if (_filterPosition == _patternSequence.Length)
1920 //{
1921 //    _filterPosition = -2; // Длиннее чем нужно
1922 //    return false;
1923 //}
1924 //if (element != _patternSequence[_filterPosition])
1925 //{
1926 //    _filterPosition = -1;
1927 //    return false; // Начинается иначе
1928 //}
1929 //_filterPosition++;
1930 //if (_filterPosition == (_patternSequence.Length - 1))
1931 //    return false;
1932 //if (_filterPosition >= 0)
1933 //{
1934 //    if (element == _patternSequence[_filterPosition + 1])
1935 //        _filterPosition++;
1936 //    else
1937 //        return false;
1938 //}
1939 //if (_filterPosition < 0)
1940 //{
1941 //    if (element == _patternSequence[0])
1942 //        _filterPosition = 0;
1943 //}
1944 }
1945
1946 public void AddAllPatternMatchedToResults(IEnumerable<ulong> sequencesToMatch)
1947 {
1948     foreach (var sequenceToMatch in sequencesToMatch)
1949     {
1950         if (PatternMatch(sequenceToMatch))
1951         {
1952             _results.Add(sequenceToMatch);
1953         }
1954     }
1955 }
1956 }
1957
1958 #endregion
1959 }
1960 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5

```

```

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

```

./Platform.Data.Doublets/Sequences/SequencesOptions.cs

```

1 using System;
2 using System.Collections.Generic;
3 using Platform.Interfaces;
4 using Platform.Collections.Stacks;
5 using Platform.Data.Doublets.Sequences.Frequencies.Cache;
6 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
7 using Platform.Data.Doublets.Sequences.Converters;
8 using Platform.Data.Doublets.Sequences.CriteriaMatchers;
9 using Platform.Data.Doublets.Sequences.Walkers;
10 using Platform.Data.Doublets.Sequences.Indexes;
11
12 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
13
14 namespace Platform.Data.Doublets.Sequences
15 {
16     public class SequencesOptions<TLink> // TODO: To use type parameter <TLink> the
            ↳ ILinks<TLink> must contain GetConstants function.
17     {
18         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↳ EqualityComparer<TLink>.Default;
19
20         public TLink SequenceMarkerLink { get; set; }
21         public bool UseCascadeUpdate { get; set; }
22         public bool UseCascadeDelete { get; set; }
23         public bool UseIndex { get; set; } // TODO: Update Index on sequence update/delete.
24         public bool UseSequenceMarker { get; set; }
25         public bool UseCompression { get; set; }
26         public bool UseGarbageCollection { get; set; }
27         public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting { get; set; }
28         public bool EnforceSingleSequenceVersionOnWriteBasedOnNew { get; set; }
29
30         public MarkedSequenceCriterionMatcher<TLink> MarkedSequenceMatcher { get; set; }
31         public IConverter<IList<TLink>, TLink> LinksToSequenceConverter { get; set; }
32         public ISequenceIndex<TLink> Index { get; set; }
33         public ISequenceWalker<TLink> Walker { get; set; }
34         public bool ReadFullSequence { get; set; }
35
36         // TODO: Реализовать компактификацию при чтении
37         //public bool EnforceSingleSequenceVersionOnRead { get; set; }
38         //public bool UseRequestMarker { get; set; }
39         //public bool StoreRequestResults { get; set; }
40
41         public void InitOptions(ISynchronizedLinks<TLink> links)
42         {
43             if (UseSequenceMarker)
44             {
45                 if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
46                 {
47                     SequenceMarkerLink = links.CreatePoint();
48                 }
49             }
50         }
51     }
52 }

```



```

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

```

./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 }

```

```

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

```

```

55         nextArray[doubletOffset] = candidate;
56     }
57     else
58     {
59         var link = Links.GetLink(candidate);
60         var linkSource = Links.GetSource(link);
61         var linkTarget = Links.GetTarget(link);
62         nextArray[doubletOffset] = linkSource;
63         nextArray[doubletOffset + 1] = linkTarget;
64         if (!hasElements)
65         {
66             hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
67         }
68     }
69 }
70 #if USEARRAYPOOL
71     if (array.Length > 1)
72     {
73         ArrayPool.Free(array);
74     }
75 #endif
76     array = nextArray;
77 }
78 while (hasElements);
79 var filledElementsCount = CountFilledElements(array);
80 if (filledElementsCount == array.Length)
81 {
82     return array;
83 }
84 else
85 {
86     return CopyFilledElements(array, filledElementsCount);
87 }
88 }
89
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 private static TLink[] CopyFilledElements(TLink[] array, int filledElementsCount)
92 {
93     var finalArray = new TLink[filledElementsCount];
94     for (int i = 0, j = 0; i < array.Length; i++)
95     {
96         if (!_equalityComparer.Equals(array[i], default))
97         {
98             finalArray[j] = array[i];
99             j++;
100         }
101     }
102     #if USEARRAYPOOL
103         ArrayPool.Free(array);
104     #endif
105     return finalArray;
106 }
107
108 [MethodImpl(MethodImplOptions.AggressiveInlining)]
109 private static int CountFilledElements(TLink[] array)
110 {
111     var count = 0;
112     for (var i = 0; i < array.Length; i++)
113     {
114         if (!_equalityComparer.Equals(array[i], default))
115         {
116             count++;
117         }
118     }
119     return count;
120 }
121 }
122 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Collections.Stacks;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Sequences.Walkers
9 {
10     public class RightSequenceWalker<TLink> : SequenceWalkerBase<TLink>

```

```

11 {
12     public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
        ↳ isElement) : base(links, stack, isElement) { }
13
14     public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links,
        ↳ stack, links.IsPartialPoint) { }
15
16     [MethodImpl(MethodImplOptions.AggressiveInlining)]
17     protected override TLink GetNextElementAfterPop(TLink element) =>
        ↳ Links.GetTarget(element);
18
19     [MethodImpl(MethodImplOptions.AggressiveInlining)]
20     protected override TLink GetNextElementAfterPush(TLink element) =>
        ↳ Links.GetSource(element);
21
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>,
        ↳ ISequenceWalker<TLink>
11     {
12         private readonly IStack<TLink> _stack;
13         private readonly Func<TLink, bool> _isElement;
14
15         protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
        ↳ isElement) : base(links)
16         {
17             _stack = stack;
18             _isElement = isElement;
19         }
20
21         protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : this(links,
        ↳ stack, links.IsPartialPoint)
22         {
23         }
24
25         public IEnumerable<TLink> Walk(TLink sequence)
26         {
27             _stack.Clear();
28             var element = sequence;
29             if (IsElement(element))
30             {
31                 yield return element;
32             }
33             else
34             {
35                 while (true)
36                 {
37                     if (IsElement(element))
38                     {
39                         if (_stack.IsEmpty)
40                         {
41                             break;
42                         }
43                         element = _stack.Pop();
44                         foreach (var output in WalkContents(element))

```

```

45         {
46             yield return output;
47         }
48         element = GetNextElementAfterPop(element);
49     }
50     else
51     {
52         _stack.Push(element);
53         element = GetNextElementAfterPush(element);
54     }
55 }
56 }
57 }
58
59 [MethodImpl(MethodImplOptions.AggressiveInlining)]
60 protected virtual bool IsElement(TLink elementLink) => _isElement(elementLink);
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 protected abstract TLink GetNextElementAfterPop(TLink element);
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 protected abstract TLink GetNextElementAfterPush(TLink element);
67
68 [MethodImpl(MethodImplOptions.AggressiveInlining)]
69 protected abstract IEnumerable<TLink> WalkContents(TLink element);
70 }
71 }

```

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

```

1  using System.Collections.Generic;
2  using Platform.Collections.Stacks;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Stacks
7  {
8      public class Stack<TLink> : IStack<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ↪ EqualityComparer<TLink>.Default;
12
13         private readonly ILinks<TLink> _links;
14         private readonly TLink _stack;
15
16         public bool IsEmpty => _equalityComparer.Equals(Peek(), _stack);
17
18         public Stack(ILinks<TLink> links, TLink stack)
19         {
20             _links = links;
21             _stack = stack;
22         }
23
24         private TLink GetStackMarker() => _links.GetSource(_stack);
25
26         private TLink GetTop() => _links.GetTarget(_stack);
27
28         public TLink Peek() => _links.GetTarget(GetTop());
29
30         public TLink Pop()
31         {
32             var element = Peek();
33             if (!_equalityComparer.Equals(element, _stack))
34             {
35                 var top = GetTop();
36                 var previousTop = _links.GetSource(top);
37                 _links.Update(_stack, GetStackMarker(), previousTop);
38                 _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
23             ↳ ReaderWriterLockSynchronization(), links) { }
24
25         public SynchronizedLinks(ISynchronization synchronization, ILinks<TLinkAddress> links)
26         {
27             SyncRoot = synchronization;
28             Sync = this;
29             Unsync = links;
30             Constants = links.Constants;
31         }
32
33         public TLinkAddress Count(IList<TLinkAddress> restriction) =>
34             ↳ SyncRoot.ExecuteReadOperation(restriction, Unsync.Count);
35         public TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler,
36             ↳ IList<TLinkAddress> restrictions) => SyncRoot.ExecuteReadOperation(handler,
37             ↳ restrictions, (handler1, restrictions1) => Unsync.Each(handler1, restrictions1));
38         public TLinkAddress Create(IList<TLinkAddress> restrictions) =>
39             ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Create);
40         public TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress>
41             ↳ substitution) => SyncRoot.ExecuteWriteOperation(restrictions, substitution,
42             ↳ Unsync.Update);
43         public void Delete(IList<TLinkAddress> restrictions) =>
44             ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Delete);
45
46         //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
47         ↳ IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
48         //{
49         //    if (restriction != null && substitution != null &&
50         ↳ !substitution.EqualTo(restriction))
51         //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
52         ↳ substitution, substitutedHandler, Unsync.Trigger);
53         //
54         //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
55         ↳ substitutedHandler, Unsync.Trigger);
56         //}
57     }
58 }

```

./Platform.Data.Doublets/UInt64Link.cs

```

1 using System;
2 using System.Collections;
3 using System.Collections.Generic;
4 using Platform.Exceptions;
5 using Platform.Ranges;

```

```

6 using Platform.Singletons;
7 using Platform.Collections.Lists;
8
9 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets
12 {
13     /// <summary>
14     /// Структура описывающая уникальную связь.
15     /// </summary>
16     public struct UInt64Link : IEquatable<UInt64Link>, IReadOnlyList<ulong>, IList<ulong>
17     {
18         private static readonly LinksConstants<ulong> _constants =
19             ↪ Default<LinksConstants<ulong>>.Instance;
20
21         private const int Length = 3;
22
23         public readonly ulong Index;
24         public readonly ulong Source;
25         public readonly ulong Target;
26
27         public static readonly UInt64Link Null = new UInt64Link();
28
29         public UInt64Link(params ulong[] values)
30         {
31             Index = values.Length > _constants.IndexPart ? values[_constants.IndexPart] :
32                 ↪ _constants.Null;
33             Source = values.Length > _constants.SourcePart ? values[_constants.SourcePart] :
34                 ↪ _constants.Null;
35             Target = values.Length > _constants.TargetPart ? values[_constants.TargetPart] :
36                 ↪ _constants.Null;
37         }
38
39         public UInt64Link(IList<ulong> values)
40         {
41             Index = values.Count > _constants.IndexPart ? values[_constants.IndexPart] :
42                 ↪ _constants.Null;
43             Source = values.Count > _constants.SourcePart ? values[_constants.SourcePart] :
44                 ↪ _constants.Null;
45             Target = values.Count > _constants.TargetPart ? values[_constants.TargetPart] :
46                 ↪ _constants.Null;
47         }
48
49         public UInt64Link(ulong index, ulong source, ulong target)
50         {
51             Index = index;
52             Source = source;
53             Target = target;
54         }
55
56         public UInt64Link(ulong source, ulong target)
57             : this(_constants.Null, source, target)
58         {
59             Source = source;
60             Target = target;
61         }
62
63         public static UInt64Link Create(ulong source, ulong target) => new UInt64Link(source,
64             ↪ target);
65
66         public override int GetHashCode() => (Index, Source, Target).GetHashCode();
67
68         public bool IsNull() => Index == _constants.Null
69             && Source == _constants.Null
70             && Target == _constants.Null;
71
72         public override bool Equals(object other) => other is UInt64Link &&
73             ↪ Equals((UInt64Link)other);
74
75         public bool Equals(UInt64Link other) => Index == other.Index
76             && Source == other.Source
77             && Target == other.Target;
78
79         public static string ToString(ulong index, ulong source, ulong target) => $"({index}:
80             ↪ {source}->{target})";
81
82         public static string ToString(ulong source, ulong target) => $"({source}->{target})";
83
84         public static implicit operator ulong[] (UInt64Link link) => link.ToArray();
85     }
86 }

```



```

76 public static implicit operator UInt64Link(ulong[] linkArray) => new
    ↳ UInt64Link(linkArray);
77
78 public override string ToString() => Index == _constants.Null ? ToString(Source, Target)
    ↳ : ToString(Index, Source, Target);
79
80 #region IList
81
82 public ulong this[int index]
83 {
84     get
85     {
86         Ensure.OnDebug.ArgumentInRange(index, new Range<int>(0, Length - 1),
            ↳ nameof(index));
87         if (index == _constants.IndexPart)
88         {
89             return Index;
90         }
91         if (index == _constants.SourcePart)
92         {
93             return Source;
94         }
95         if (index == _constants.TargetPart)
96         {
97             return Target;
98         }
99         throw new NotSupportedException(); // Impossible path due to
            ↳ Ensure.ArgumentInRange
100     }
101     set => throw new NotSupportedException();
102 }
103
104 public int Count => Length;
105
106 public bool IsReadOnly => true;
107
108 IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
109
110 public IEnumerator<ulong> GetEnumerator()
111 {
112     yield return Index;
113     yield return Source;
114     yield return Target;
115 }
116
117 public void Add(ulong item) => throw new NotSupportedException();
118
119 public void Clear() => throw new NotSupportedException();
120
121 public bool Contains(ulong item) => IndexOf(item) >= 0;
122
123 public void CopyTo(ulong[] array, int arrayIndex)
124 {
125     Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
126     Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
        ↳ nameof(arrayIndex));
127     if (arrayIndex + Length > array.Length)
128     {
129         throw new ArgumentException();
130     }
131     array[arrayIndex++] = Index;
132     array[arrayIndex++] = Source;
133     array[arrayIndex] = Target;
134 }
135
136 public bool Remove(ulong item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
137
138 public int IndexOf(ulong item)
139 {
140     if (Index == item)
141     {
142         return _constants.IndexPart;
143     }
144     if (Source == item)
145     {
146         return _constants.SourcePart;
147     }
148     if (Target == item)
149     {

```

```

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

```

./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         }
51     }
52 }

```

```

48 public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
49 {
50     if (sequence == null)
51     {
52         return false;
53     }
54     var constants = links.Constants;
55     for (var i = 0; i < sequence.Length; i++)
56     {
57         if (sequence[i] == constants.Any)
58         {
59             return true;
60         }
61     }
62     return false;
63 }
64
65 public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
66     ↪ Func<UInt64Link, bool> isElement, bool renderIndex = false, bool renderDebug = false)
67 {
68     var sb = new StringBuilder();
69     var visited = new HashSet<ulong>();
70     links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
71     ↪ innerSb.Append(link.Index), renderIndex, renderDebug);
72     return sb.ToString();
73 }
74
75 public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
76     ↪ Func<UInt64Link, bool> isElement, Action<StringBuilder, UInt64Link> appendElement,
77     ↪ bool renderIndex = false, bool renderDebug = false)
78 {
79     var sb = new StringBuilder();
80     var visited = new HashSet<ulong>();
81     links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
82     ↪ renderDebug);
83     return sb.ToString();
84 }
85
86 public static void AppendStructure(this ILinks<ulong> links, StringBuilder sb,
87     ↪ HashSet<ulong> visited, ulong linkIndex, Func<UInt64Link, bool> isElement,
88     ↪ Action<StringBuilder, UInt64Link> appendElement, bool renderIndex = false, bool
89     ↪ renderDebug = false)
90 {
91     if (sb == null)
92     {
93         throw new ArgumentNullException(nameof(sb));
94     }
95     if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
96     ↪ Constants.Itself)
97     {
98         return;
99     }
100     if (links.Exists(linkIndex))
101     {
102         if (visited.Add(linkIndex))
103         {
104             sb.Append('(');
105             var link = new UInt64Link(links.GetLink(linkIndex));
106             if (renderIndex)
107             {
108                 sb.Append(link.Index);
109                 sb.Append(':');
110             }
111             if (link.Source == link.Index)
112             {
113                 sb.Append(link.Index);
114             }
115             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         }
129     }
130 }

```

```

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

```

./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs

```

1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4  using System.IO;
5  using System.Runtime.CompilerServices;
6  using System.Threading;
7  using System.Threading.Tasks;
8  using Platform.Disposables;
9  using Platform.Timestamps;
10 using Platform.Unsafe;
11 using Platform.IO;
12 using Platform.Data.Doublets.Decorators;
13 using Platform.Exceptions;
14
15 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
16
17 namespace Platform.Data.Doublets
18 {
19     public class UInt64LinksTransactionsLayer : LinksDisposableDecoratorBase //-V3073
20     {
21         /// <remarks>
22         /// Альтернативные варианты хранения трансформации (элемента транзакции):
23         ///
24         /// private enum TransitionType
25         /// {
26         ///     Creation,
27         ///     UpdateOf,
28         ///     UpdateTo,
29         ///     Deletion
30         /// }
31         ///
32         /// private struct Transition
33         /// {
34         ///     public ulong TransactionId;
35         ///     public UniqueTimestamp Timestamp;

```

```

36     ///     public TransactionItemType Type;
37     ///     public Link Source;
38     ///     public Link Linker;
39     ///     public Link Target;
40     /// }
41     ///
42     /// Или
43     ///
44     /// public struct TransitionHeader
45     /// {
46     ///     public ulong TransactionIdCombined;
47     ///     public ulong TimestampCombined;
48     ///
49     ///     public ulong TransactionId
50     ///     {
51     ///         get
52     ///         {
53     ///             return (ulong) mask & TransactionIdCombined;
54     ///         }
55     ///     }
56     ///
57     ///     public UniqueTimestamp Timestamp
58     ///     {
59     ///         get
60     ///         {
61     ///             return (UniqueTimestamp)mask & TransactionIdCombined;
62     ///         }
63     ///     }
64     ///
65     ///     public TransactionItemType Type
66     ///     {
67     ///         get
68     ///         {
69     ///             // Использовать по одному биту из TransactionId и Timestamp,
70     ///             // для значения в 2 бита, которое представляет тип операции
71     ///             throw new NotImplementedException();
72     ///         }
73     ///     }
74     /// }
75     ///
76     /// private struct Transition
77     /// {
78     ///     public TransitionHeader Header;
79     ///     public Link Source;
80     ///     public Link Linker;
81     ///     public Link Target;
82     /// }
83     ///
84     /// </remarks>
85 public struct Transition
86 {
87     public static readonly long Size = Structure<Transition>.Size;
88
89     public readonly ulong TransactionId;
90     public readonly UInt64Link Before;
91     public readonly UInt64Link After;
92     public readonly Timestamp Timestamp;
93
94     public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
↵ transactionId, UInt64Link before, UInt64Link after)
95     {
96         TransactionId = transactionId;
97         Before = before;
98         After = after;
99         Timestamp = uniqueTimestampFactory.Create();
100     }
101
102     public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
↵ transactionId, UInt64Link before)
103         : this(uniqueTimestampFactory, transactionId, before, default)
104     {
105     }
106
107     public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId)
108         : this(uniqueTimestampFactory, transactionId, default, default)
109     {
110     }
111

```

```

112     public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
113         ↳ {After}";
114 }
115
116 /// <remarks>
117 /// Другие варианты реализации транзакций (атомарности):
118 /// 1. Разделение хранения значения связи ((Source Target) или (Source Linker
119 ↳ Target)) и индексов.
120 /// 2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
121 ↳ потребуется решить вопрос
122 ↳ со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
123 ↳ пересечениями идентификаторов.
124 ///
125 /// Где хранить промежуточный список транзакций?
126 ///
127 /// В оперативной памяти:
128 /// Минусы:
129 /// 1. Может усложнить систему, если она будет функционировать самостоятельно,
130 /// так как нужно отдельно выделять память под список трансформаций.
131 /// 2. Выделенной оперативной памяти может не хватить, в том случае,
132 /// если транзакция использует слишком много трансформаций.
133 ↳ Можно использовать жёсткий диск для слишком длинных транзакций.
134 ↳ Максимальный размер списка трансформаций можно ограничить / задать
135 ↳ константой.
136 /// 3. При подтверждении транзакции (Commit) все трансформации записываются разом
137 ↳ создавая задержку.
138 ///
139 /// На жёстком диске:
140 /// Минусы:
141 /// 1. Длительный отклик, на запись каждой трансформации.
142 /// 2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
143 ↳ Это может решаться упаковкой/исключением дублирующих операций.
144 ↳ Также это может решаться тем, что короткие транзакции вообще
145 ↳ не будут записываться в случае отката.
146 /// 3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
147 ↳ операции (трансформации)
148 ↳ будут записаны в лог.
149 ///
150 /// </remarks>
151 public class Transaction : DisposableBase
152 {
153     private readonly Queue<Transition> _transitions;
154     private readonly UInt64LinksTransactionsLayer _layer;
155     public bool IsCommitted { get; private set; }
156     public bool IsReverted { get; private set; }
157
158     public Transaction(UInt64LinksTransactionsLayer layer)
159     {
160         _layer = layer;
161         if (_layer._currentTransactionId != 0)
162         {
163             throw new NotSupportedException("Nested transactions not supported.");
164         }
165         IsCommitted = false;
166         IsReverted = false;
167         _transitions = new Queue<Transition>();
168         SetCurrentTransaction(layer, this);
169     }
170
171     public void Commit()
172     {
173         EnsureTransactionAllowsWriteOperations(this);
174         while (_transitions.Count > 0)
175         {
176             var transition = _transitions.Dequeue();
177             _layer._transitions.Enqueue(transition);
178         }
179         _layer._lastCommittedTransactionId = _layer._currentTransactionId;
180         IsCommitted = true;
181     }
182
183     private void Revert()
184     {
185         EnsureTransactionAllowsWriteOperations(this);
186         var transitionsToRevert = new Transition[_transitions.Count];
187         _transitions.CopyTo(transitionsToRevert, 0);
188         for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
189         {

```

```

183         _layer.RevertTransition(transitionsToRevert[i]);
184     }
185     IsReverted = true;
186 }
187
188 public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
189     ↪ Transaction transaction)
190 {
191     layer._currentTransactionId = layer._lastCommittedTransactionId + 1;
192     layer._currentTransactionTransitions = transaction._transitions;
193     layer._currentTransaction = transaction;
194 }
195
196 public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
197 {
198     if (transaction.IsReverted)
199     {
200         throw new InvalidOperationException("Transation is reverted.");
201     }
202     if (transaction.IsCommitted)
203     {
204         throw new InvalidOperationException("Transation is committed.");
205     }
206 }
207
208 protected override void Dispose(bool manual, bool wasDisposed)
209 {
210     if (!wasDisposed && _layer != null && !_layer.IsDisposed)
211     {
212         if (!IsCommitted && !IsReverted)
213         {
214             Revert();
215         }
216         _layer.ResetCurrentTransation();
217     }
218 }
219
220 public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
221
222 private readonly string _logAddress;
223 private readonly FileStream _log;
224 private readonly Queue<Transition> _transitions;
225 private readonly UniqueTimestampFactory _uniqueTimestampFactory;
226 private Task _transitionsPusher;
227 private Transition _lastCommittedTransition;
228 private ulong _currentTransactionId;
229 private Queue<Transition> _currentTransactionTransitions;
230 private Transaction _currentTransaction;
231 private ulong _lastCommittedTransactionId;
232
233 public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
234     : base(links)
235 {
236     if (string.IsNullOrEmpty(logAddress))
237     {
238         throw new ArgumentNullException(nameof(logAddress));
239     }
240     // В первой строке файла хранится последняя закоммиченную транзакцию.
241     // При запуске это используется для проверки удачного закрытия файла лога.
242     // In the first line of the file the last committed transaction is stored.
243     // On startup, this is used to check that the log file is successfully closed.
244     var lastCommittedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
245     var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
246     if (!lastCommittedTransition.Equals(lastWrittenTransition))
247     {
248         Dispose();
249         throw new NotSupportedException("Database is damaged, autorecovery is not
250             ↪ supported yet.");
251     }
252     if (lastCommittedTransition.Equals(default(Transition)))
253     {
254         FileHelpers.WriteFirst(logAddress, lastCommittedTransition);
255     }
256     _lastCommittedTransition = lastCommittedTransition;
257     // TODO: Think about a better way to calculate or store this value
258     var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
259     _lastCommittedTransactionId = allTransitions.Max(x => x.TransactionId);
260     _uniqueTimestampFactory = new UniqueTimestampFactory();

```

```

260     _logAddress = logAddress;
261     _log = FileHelpers.Append(logAddress);
262     _transitions = new Queue<Transition>();
263     _transitionsPusher = new Task(TransitionsPusher);
264     _transitionsPusher.Start();
265 }
266
267 public IList<ulong> GetLinkValue(ulong link) => Links.GetLink(link);
268
269 public override ulong Create(IList<ulong> restrictions)
270 {
271     var createdLinkIndex = Links.Create();
272     var createdLink = new UInt64Link(Links.GetLink(createdLinkIndex));
273     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
274     ↪ default, createdLink));
275     return createdLinkIndex;
276 }
277
278 public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
279 {
280     var linkIndex = restrictions[Constants.IndexPart];
281     var beforeLink = new UInt64Link(Links.GetLink(linkIndex));
282     linkIndex = Links.Update(restrictions, substitution);
283     var afterLink = new UInt64Link(Links.GetLink(linkIndex));
284     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
285     ↪ beforeLink, afterLink));
286     return linkIndex;
287 }
288
289 public override void Delete(IList<ulong> restrictions)
290 {
291     var link = restrictions[Constants.IndexPart];
292     var deletedLink = new UInt64Link(Links.GetLink(link));
293     Links.Delete(link);
294     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
295     ↪ deletedLink, default));
296 }
297
298 [MethodImpl(MethodImplOptions.AggressiveInlining)]
299 private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
300 ↪ _transitions;
301
302 private void CommitTransition(Transition transition)
303 {
304     if (_currentTransaction != null)
305     {
306         Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
307     }
308     var transitions = GetCurrentTransitions();
309     transitions.Enqueue(transition);
310 }
311
312 private void RevertTransition(Transition transition)
313 {
314     if (transition.After.IsNull()) // Revert Deletion with Creation
315     {
316         Links.Create();
317     }
318     else if (transition.Before.IsNull()) // Revert Creation with Deletion
319     {
320         Links.Delete(transition.After.Index);
321     }
322     else // Revert Update
323     {
324         Links.Update(new[] { transition.After.Index, transition.Before.Source,
325         ↪ transition.Before.Target });
326     }
327 }
328
329 private void ResetCurrentTransation()
330 {
331     _currentTransactionId = 0;
332     _currentTransactionTransitions = null;
333     _currentTransaction = null;
334 }
335
336 private void PushTransitions()
337 {

```



```

333         if (_log == null || _transitions == null)
334         {
335             return;
336         }
337         for (var i = 0; i < _transitions.Count; i++)
338         {
339             var transition = _transitions.Dequeue();
340
341             _log.Write(transition);
342             _lastCommittedTransition = transition;
343         }
344     }
345
346     private void TransitionsPusher()
347     {
348         while (!IsDisposed && _transitionsPusher != null)
349         {
350             Thread.Sleep(DefaultPushDelay);
351             PushTransitions();
352         }
353     }
354
355     public Transaction BeginTransaction() => new Transaction(this);
356
357     private void DisposeTransitions()
358     {
359         try
360         {
361             var pusher = _transitionsPusher;
362             if (pusher != null)
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         }
19     }
20

```

```

16         _unicodeSymbolMarker = unicodeSymbolMarker;
17     }
18
19     public TLink Convert(char source)
20     {
21         var unaryNumber = _addressToNumberConverter.Convert((Integer<TLink>)source);
22         return Links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
23     }
24 }
25 }

```

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

```

1 using Platform.Data.Doublets.Sequences.Indexes;
2 using Platform.Interfaces;
3 using System.Collections.Generic;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Unicode
8 {
9     public class StringToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
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 }

```

```

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];

```

```

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

```

```

178         var maxLength = offset + relativeLength;
179         for (var i = offset; i < maxLength; i++)
180         {
181             innerSequence[i - offset] = array[i];
182         }
183         result.Add(innerSequence);
184         offset += relativeLength;
185     }
186     return result;
187 }
188 }
189 }

```

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

```

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

```

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

```

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

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using Xunit;
4  using Platform.Diagnostics;
5
6  namespace Platform.Data.Doublets.Tests
7  {
8      public static class ComparisonTests
9      {
10         private class UInt64Comparer : IComparer<ulong>
11         {
12             public int Compare(ulong x, ulong y) => x.CompareTo(y);
13         }
14
15         private static int Compare(ulong x, ulong y) => x.CompareTo(y);
16
17         [Fact]
18         public static void GreaterOrEqualPerformanceTest()
19         {
20             const int N = 1000000;
21
22             ulong x = 10;
23             ulong y = 500;
24
25             bool result = false;
26

```

```

27     var ts1 = Performance.Measure(() =>
28     {
29         for (int i = 0; i < N; i++)
30         {
31             result = Compare(x, y) >= 0;
32         }
33     });
34
35     var comparer1 = Comparer<ulong>.Default;
36
37     var ts2 = Performance.Measure(() =>
38     {
39         for (int i = 0; i < N; i++)
40         {
41             result = comparer1.Compare(x, y) >= 0;
42         }
43     });
44
45     Func<ulong, ulong, int> compareReference = comparer1.Compare;
46
47     var ts3 = Performance.Measure(() =>
48     {
49         for (int i = 0; i < N; i++)
50         {
51             result = compareReference(x, y) >= 0;
52         }
53     });
54
55     var comparer2 = new UInt64Comparer();
56
57     var ts4 = Performance.Measure(() =>
58     {
59         for (int i = 0; i < N; i++)
60         {
61             result = comparer2.Compare(x, y) >= 0;
62         }
63     });
64
65     Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {result}");
66 }
67 }
68 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using Xunit;
4  using Platform.Diagnostics;
5
6  namespace Platform.Data.Doublets.Tests
7  {
8      public static class EqualityTests
9      {
10         protected class UInt64EqualityComparer : IEqualityComparer<ulong>
11         {
12             public bool Equals(ulong x, ulong y) => x == y;
13
14             public int GetHashCode(ulong obj) => obj.GetHashCode();
15         }
16
17         private static bool Equals1<T>(T x, T y) => Equals(x, y);
18
19         private static bool Equals2<T>(T x, T y) => x.Equals(y);
20
21         private static bool Equals3(ulong x, ulong y) => x == y;
22
23         [Fact]
24         public static void 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         Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
34             ↪ MultipleRandomCreationsAndDeletions(16)); // Cannot use more because current
35             ↪ implementation of tree cuts out 5 bits from the address space.
36         Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Te
37             ↪ stMultipleRandomCreationsAndDeletions(100));
38         Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
39             ↪ MultipleRandomCreationsAndDeletions(100));
40         Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
41             ↪ tMultipleRandomCreationsAndDeletions(100));
42     }
43
44     private static void Using<TLink>(Action<ILinks<TLink>> action)
45     {
46         using (var scope = new Scope<Types<HeapResizableDirectMemory,
47             ↪ ResizableDirectMemoryLinks<TLink>>>())
48         {
49             action(scope.Use<ILinks<TLink>>());
50         }
51     }
52 }
53 }

```

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

```

1 using System;
2 using System.Linq;
3 using System.Collections.Generic;
4 using Xunit;
5 using Platform.Data.Doublets.Sequences;
6 using Platform.Data.Doublets.Sequences.Frequencies.Cache;
7 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
8 using Platform.Data.Doublets.Sequences.Converters;
9 using Platform.Data.Doublets.PropertyOperators;
10 using Platform.Data.Doublets.Incrementers;
11 using Platform.Data.Doublets.Sequences.Walkers;
12 using Platform.Data.Doublets.Sequences.Indexes;
13 using Platform.Data.Doublets.Unicode;
14 using Platform.Data.Doublets.Numbers.Unary;
15
16 namespace Platform.Data.Doublets.Tests
17 {
18     public static class OptimalVariantSequenceTests
19     {
20         private const string SequenceExample = "зеленела зелёная зелень";
21
22         [Fact]
23         public static void LinksBasedFrequencyStoredOptimalVariantSequenceTest()
24         {
25             using (var scope = new TempLinksTestScope(useSequences: false))
26             {
27                 var links = scope.Links;
28                 var constants = links.Constants;
29
30                 links.UseUnicode();
31
32                 var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);
33
34                 var meaningRoot = links.CreatePoint();

```

```

35     var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
36     var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
37     var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
38         ↪ constants.Itself);
39
40     var unaryNumberToAddressConverter = new
41         ↪ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
42     var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
43     var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
44         ↪ frequencyMarker, unaryOne, unaryNumberIncrementer);
45     var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
46         ↪ frequencyPropertyMarker, frequencyMarker);
47     var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
48         ↪ frequencyPropertyOperator, frequencyIncrementer);
49     var linkToItsFrequencyNumberConverter = new
50         ↪ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
51         ↪ unaryNumberToAddressConverter);
52     var sequenceToItsLocalElementLevelsConverter = new
53         ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
54         ↪ linkToItsFrequencyNumberConverter);
55     var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
56         ↪ sequenceToItsLocalElementLevelsConverter);
57
58     var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
59         ↪ Walker = new LeveledSequenceWalker<ulong>(links) });
60
61     ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
62         ↪ index, optimalVariantConverter);
63 }
64
65 [Fact]
66 public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
67 {
68     using (var scope = new TempLinksTestScope(useSequences: false))
69     {
70         var links = scope.Links;
71
72         links.UseUnicode();
73
74         var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);
75
76         var linksToFrequencies = new Dictionary<ulong, ulong>();
77
78         var totalSequenceSymbolFrequencyCounter = new
79             ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links);
80
81         var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
82             ↪ totalSequenceSymbolFrequencyCounter);
83
84         var index = new
85             ↪ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
86         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache);
87
88         var sequenceToItsLocalElementLevelsConverter = new
89             ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
90             ↪ linkToItsFrequencyNumberConverter);
91         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
92             ↪ sequenceToItsLocalElementLevelsConverter);
93
94         var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
95             ↪ Walker = new LeveledSequenceWalker<ulong>(links) });
96
97         ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
98             ↪ index, optimalVariantConverter);
99     }
100 }
101
102 private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
103     ↪ SequenceToItsLocalElementLevelsConverter<ulong>
104     ↪ sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
105     ↪ OptimalVariantConverter<ulong> optimalVariantConverter)
106 {
107     index.Add(sequence);
108
109     var optimalVariant = optimalVariantConverter.Convert(sequence);
110 }

```

```

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        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            private static void TestBasicMemoryOperations(this ILinks<ulong> memoryAdapter)
37            {
38                var link = memoryAdapter.Create();
39                memoryAdapter.Delete(link);
40            }
41
42            [Fact]
43            public static void NonexistentReferencesHeapMemoryTest()
44            {
45                using (var memory = new
46                    ↳ HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
47                using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
48                    ↳ UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
49                {
50                    memoryAdapter.TestNonexistentReferences();
51                }
52
53                private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
54                {
55                    var link = memoryAdapter.Create();
56                    memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
57                    var resultLink = _constants.Null;
58                    memoryAdapter.Each(foundLink =>
59                    {
60                        resultLink = foundLink[_constants.IndexPart];
61                        return _constants.Break;
62                    }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
63                    Assert.True(resultLink == link);
64                    Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
65                    memoryAdapter.Delete(link);
66                }
67            }
68        }
69    }
70 }

```

./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 performance measurements
26             _ = BitString.GetBitMaskFromIndex(1);
27         }
28
29         [Fact]
30         public static void CreateAllVariantsTest()
31         {
32             const long sequenceLength = 8;
33
34             using (var scope = new TempLinksTestScope(useSequences: true))

```

```

34     {
35         var links = scope.Links;
36         var sequences = scope.Sequences;
37
38         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 //
81 //        // TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
82 //        o.
83 //
84 //        var sequences = new Sequences(links);
85 //
86 //        var sw1 = Stopwatch.StartNew();
87 //        var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
88 //
89 //        var sw2 = Stopwatch.StartNew();
90 //        var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
91 //
92 //        Assert.True(results1.Count > results2.Length);
93 //        Assert.True(sw1.Elapsed > sw2.Elapsed);
94 //
95 //        for (var i = 0; i < sequenceLength; i++)
96 //            links.Delete(sequence[i]);
97 //    }
98 //
99 //    File.Delete(tempFilename);
100 //}
101
102 [Fact]
103 public static void AllVariantsSearchTest()
104 {
105     const long sequenceLength = 8;
106
107     using (var scope = new TempLinksTestScope(useSequences: true))
108     {
109         var links = scope.Links;
110         var sequences = scope.Sequences;
111
112         var sequence = new ulong[sequenceLength];
113         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());

```

```

191         //Assert.True(sw1.Elapsed < sw2.Elapsed);
192
193     for (var i = 0; i < sequenceLength; i++)
194     {
195         links.Delete(sequence[i]);
196     }
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         Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
225
226         var sw1 = Stopwatch.StartNew();
227         var searchResults1 =
228         ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
229
230         var sw2 = Stopwatch.StartNew();
231         var searchResults2 =
232         ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
233
234         //var sw3 = Stopwatch.StartNew();
235         //var searchResults3 =
236         ↪ sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
237
238         var sw4 = Stopwatch.StartNew();
239         var searchResults4 =
240         ↪ sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();
241
242         //Global.Trash = searchResults3;
243
244         //var searchResults1Strings = searchResults1.Select(x => x + ": " +
245         ↪ sequences.FormatSequence(x)).ToList();
246         //Global.Trash = searchResults1Strings;
247
248         var intersection1 = createResults.Intersect(searchResults1).ToList();
249         Assert.True(intersection1.Count == createResults.Length);
250
251         var intersection2 = createResults.Intersect(searchResults2).ToList();
252         Assert.True(intersection2.Count == createResults.Length);
253
254         var intersection4 = createResults.Intersect(searchResults4).ToList();
255         Assert.True(intersection4.Count == createResults.Length);
256
257         for (var i = 0; i < sequenceLength; i++)
258         {
259             links.Delete(sequence[i]);
260         }
261     }
262 }
263
264 [Fact]
265 public static void BalancedPartialVariantsSearchTest()
266 {
267     const long sequenceLength = 200;

```



```

264 using (var scope = new TempLinksTestScope(useSequences: true))
265 {
266     var links = scope.Links;
267     var sequences = scope.Sequences;
268
269     var sequence = new ulong[sequenceLength];
270     for (var i = 0; i < sequenceLength; i++)
271     {
272         sequence[i] = links.Create();
273     }
274
275     var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
276
277     var balancedVariant = balancedVariantConverter.Convert(sequence);
278
279     var partialSequence = new ulong[sequenceLength - 2];
280
281     Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
282
283     var sw1 = Stopwatch.StartNew();
284     var searchResults1 =
285         ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
286
287     var sw2 = Stopwatch.StartNew();
288     var searchResults2 =
289         ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
290
291     Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
292
293     Assert.True(searchResults2.Count == 1 && balancedVariant ==
294         ↪ searchResults2.First());
295
296     for (var i = 0; i < sequenceLength; i++)
297     {
298         links.Delete(sequence[i]);
299     }
300 }
301
302 [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);

```

```

341         var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
342
343         Assert.Contains(douplet, matchedSequences4);
344         Assert.Contains(balancedVariant, matchedSequences4);
345
346         for (var i = 0; i < sequence.Length; i++)
347         {
348             links.Delete(sequence[i]);
349         }
350     }
351 }
352
353 [Fact]
354 public static void IndexTest()
355 {
356     using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
357         ↪ true }, useSequences: true))
358     {
359         var links = scope.Links;
360         var sequences = scope.Sequences;
361         var index = sequences.Options.Index;
362
363         var e1 = links.Create();
364         var e2 = links.Create();
365
366         var sequence = new[]
367         {
368             e1, e2, e1, e2 // mama / papa
369         };
370
371         Assert.False(index.MightContain(sequence));
372
373         index.Add(sequence);
374
375         Assert.True(index.MightContain(sequence));
376     }
377
378     /// <summary>Imported from https://raw.githubusercontent.com/wiki/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))
384
385     Обозначение пустоты, какое оно? Темнота ли это? Там где отсутствие света, отсутствие фотонов
386     ↪ (носителей света)? Или это то, что полностью отражает свет? Пустой белый лист бумаги? Там
387     ↪ где есть место для нового начала? Разве пустота это не характеристика пространства?
388     ↪ Пространство это то, что можно чем-то наполнить?
389
390     [[чёрное пространство, белое
391     ↪ пространство] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png
392     ↪ "чёрное пространство, белое пространство")] (https://raw.githubusercontent.com/Konard/Links
393     ↪ Platform/master/doc/Intro/1.png)
394
395     Что может быть минимальным рисунком, образом, графикой? Может быть это точка? Это ли простейшая
396     ↪ форма? Но есть ли у точки размер? Цвет? Масса? Координаты? Время существования?
397
398     [[чёрное пространство, чёрная
399     ↪ точка] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png
400     ↪ "чёрное пространство, чёрная
401     ↪ точка")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png)
402
403     А что если повторить? Сделать копию? Создать дубликат? Из одного сделать два? Может это быть
404     ↪ так? Инверсия? Отражение? Сумма?
405
406     [[белая точка, чёрная
407     ↪ точка] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png "белая
408     ↪ точка, чёрная
409     ↪ точка")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png)
410
411     А что если мы вообразим движение? Нужно ли время? Каким самым коротким будет путь? Что будет
412     ↪ если этот путь зафиксировать? Запомнить след? Как две точки становятся линией? Чертой?
413     ↪ Гранью? Разделителем? Единицей?
414
415     [[две белые точки, чёрная вертикальная
416     ↪ линия] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png "две
417     ↪ белые точки, чёрная вертикальная
418     ↪ линия")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png)

```

397
398 Можно ли замкнуть движение? Может ли это быть кругом? Можно ли замкнуть время? Или остаётся
→ только спираль? Но что если замкнуть предел? Создать ограничение, разделение? Получится
→ замкнутая область? Полностью отделённая от всего остального? Но что это всё остальное? Что
→ можно делить? В каком направлении? Ничего или всё? Пустота или полнота? Начало или конец?
→ Или может быть это единица и ноль? Дуальность? Противоположность? А что будет с кругом если
→ у него нет размера? Будет ли круг точкой? Точка состоящая из точек?

399
400 [![белая вертикальная линия, чёрный
→ круг](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png "белая
→ вертикальная линия, чёрный
→ круг")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png)

401
402 Как ещё можно использовать грань, черту, линию? А что если она может что-то соединять, может
→ тогда её нужно повернуть? Почему то, что перпендикулярно вертикальному горизонтально?
→ Горизонт? Инвертирует ли это смысл? Что такое смысл? Из чего состоит смысл? Существует ли
→ элементарная единица смысла?

403
404 [![белый круг, чёрная горизонтальная
→ линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png "белый
→ круг, чёрная горизонтальная
→ линия")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png)

405
406 Соединять, допустим, а какой смысл в этом есть ещё? Что если помимо смысла "соединить,
→ связать", есть ещё и смысл направления "от начала к концу"? От предка к потомку? От
→ родителя к ребёнку? От общего к частному?

407
408 [![белая горизонтальная линия, чёрная горизонтальная
→ стрелка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png
→ "белая горизонтальная линия, чёрная горизонтальная
→ стрелка")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png)

409
410 Шаг назад. Возьмём опять отделённую область, которая лишь та же замкнутая линия, что ещё она
→ может представлять собой? Объект? Но в чём его суть? Разве не в том, что у него есть
→ граница, разделяющая внутреннее и внешнее? Допустим связь, стрелка, линия соединяет два
→ объекта, как бы это выглядело?

411
412 [![белая связь, чёрная направленная
→ связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png "белая
→ связь, чёрная направленная
→ связь")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png)

413
414 Допустим у нас есть смысл "связать" и смысл "направления", много ли это нам даёт? Много ли
→ вариантов интерпретации? А что если уточнить, каким именно образом выполнена связь? Что если
→ можно задать ей чёткий, конкретный смысл? Что это будет? Тип? Глагол? Связка? Действие?
→ Трансформация? Переход из состояния в состояние? Или всё это и есть объект, суть которого в
→ его конечном состоянии, если конечно конец определён направлением?

415
416 [![белая обычная и направленная связи, чёрная типизированная
→ связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png "белая
→ обычная и направленная связи, чёрная типизированная
→ связь")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png)

417
418 А что если всё это время, мы смотрели на суть как бы снаружи? Можно ли взглянуть на это изнутри?
→ Что будет внутри объектов? Объекты ли это? Или это связи? Может ли эта структура описать
→ сама себя? Но что тогда получится, разве это не рекурсия? Может это фрактал?

419
420 [![белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная типизированная
→ связь с рекурсивной внутренней
→ структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png
→ "белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная
→ типизированная связь с рекурсивной внутренней структурой")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png)

421
422 На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом
→ рекурсии или фрактала?

423
424 [![белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная
→ типизированная связь с двойной рекурсивной внутренней
→ структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png
→ "белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная
→ типизированная связь с двойной рекурсивной внутренней структурой")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png)

425
426 Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы? Буквы?
→ Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?

427

```

428  [![белая обычная и направленная связи со структурой из 8 цветных элементов последовательности,
    ↪   чёрная типизированная связь со структурой из 8 цветных элементов последовательности](https://
    ↪   /raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png "белая обычная и
    ↪   направленная связи со структурой из 8 цветных элементов последовательности, чёрная
    ↪   типизированная связь со структурой из 8 цветных элементов последовательности")](https://raw
    ↪   .githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png)

429
430  ...
431
432  [![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-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.Reflection;
15 using Platform.Singletons;
16 using Platform.Scopes;
17 using Platform.Counters;
18 using Platform.Diagnostics;
19 using Platform.Memory;
20 using Platform.Data.Doublets.ResizableDirectMemory;
21 using Platform.Data.Doublets.Decorators;
22
23 namespace Platform.Data.Doublets.Tests
24 {
25     public static class UInt64LinksTests
26     {
27         private static readonly LinksConstants<ulong> _constants =
28             ↪ Default<LinksConstants<ulong>>.Instance;
29
30         private const long Iterations = 10 * 1024;
31
32         #region Concept
33
34         [Fact]
35         public static void MultipleCreateAndDeleteTest()
36         {
37             using (var scope = new Scope<Types<HeapResizableDirectMemory,
38                 ↪ UInt64ResizableDirectMemoryLinks>>())
39             {
40                 new UInt64Links(scope.Use<ILinks<ulong>>()).TestMultipleRandomCreationsAndDeletions(100);
41             }
42         }
43
44         [Fact]
45         public static void CascadeUpdateTest()
46         {
47             var itself = _constants.Itself;
48
49             using (var scope = new TempLinksTestScope(useLog: true))

```

```

48     {
49         var links = scope.Links;
50
51         var l1 = links.Create();
52         var l2 = links.Create();
53
54         l2 = links.Update(l2, l2, l1, l2);
55
56         links.CreateAndUpdate(l2, itself);
57         links.CreateAndUpdate(l2, itself);
58
59         l2 = links.Update(l2, l1);
60
61         links.Delete(l2);
62
63         Global.Trash = links.Count();
64
65         links.Unsync.DisposeIfPossible(); // Close links to access log
66
67         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope
        ↪ e.TempTransactionLogFilename);
68     }
69 }
70
71 [Fact]
72 public static void BasicTransactionLogTest()
73 {
74     using (var scope = new TempLinksTestScope(useLog: true))
75     {
76         var links = scope.Links;
77         var l1 = links.Create();
78         var l2 = links.Create();
79
80         Global.Trash = links.Update(l2, l2, l1, l2);
81
82         links.Delete(l1);
83
84         links.Unsync.DisposeIfPossible(); // Close links to access log
85
86         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope
        ↪ e.TempTransactionLogFilename);
87     }
88 }
89
90 [Fact]
91 public static void TransactionAutoRevertedTest()
92 {
93     // Auto Reverted (Because no commit at transaction)
94     using (var scope = new TempLinksTestScope(useLog: true))
95     {
96         var links = scope.Links;
97         var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
98         using (var transaction = transactionsLayer.BeginTransaction())
99         {
100             var l1 = links.Create();
101             var l2 = links.Create();
102
103             links.Update(l2, l2, l1, l2);
104         }
105
106         Assert.Equal(0UL, links.Count());
107
108         links.Unsync.DisposeIfPossible();
109
110         var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope
        ↪ e.TempTransactionLogFilename);
111         Assert.Single(transitions);
112     }
113 }
114
115 [Fact]
116 public static void TransactionUserCodeErrorNoDataSavedTest()
117 {
118     // User Code Error (Autoreverted), no data saved
119     var itself = _constants.Itself;
120
121     TempLinksTestScope lastScope = null;
122     try
123     {

```



```

124     using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
125         ↪ useLog: true))
126     {
127         var links = scope.Links;
128         var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor_
129         ↪ atorBase<ulong>)links.Unsync).Links;
130         using (var transaction = transactionsLayer.BeginTransaction())
131         {
132             var l1 = links.CreateAndUpdate(itself, itself);
133             var l2 = links.CreateAndUpdate(itself, itself);
134
135             l2 = links.Update(l2, l2, l1, l2);
136
137             links.CreateAndUpdate(l2, itself);
138             links.CreateAndUpdate(l2, itself);
139
140             //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi_
141             ↪ tion>(scope.TempTransactionLogFilename);
142
143             l2 = links.Update(l2, l1);
144
145             links.Delete(l2);
146
147             ExceptionThrower();
148
149             transaction.Commit();
150         }
151         Global.Trash = links.Count();
152     }
153 }
154 catch
155 {
156     Assert.False(lastScope == null);
157
158     var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l_
159     ↪ astScope.TempTransactionLogFilename);
160
161     Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
162     ↪ transitions[0].After.IsNull());
163
164     lastScope.DeleteFiles();
165 }
166 }
167
168 [Fact]
169 public static void TransactionUserCodeErrorSomeDataSavedTest()
170 {
171     // User Code Error (Autoreverted), some data saved
172     var itself = _constants.Itself;
173
174     TempLinksTestScope lastScope = null;
175     try
176     {
177         ulong l1;
178         ulong l2;
179
180         using (var scope = new TempLinksTestScope(useLog: true))
181         {
182             var links = scope.Links;
183             l1 = links.CreateAndUpdate(itself, itself);
184             l2 = links.CreateAndUpdate(itself, itself);
185
186             l2 = links.Update(l2, l2, l1, l2);
187
188             links.CreateAndUpdate(l2, itself);
189             links.CreateAndUpdate(l2, itself);
190
191             links.Unsync.DisposeIfPossible();
192
193             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(_
194             ↪ scope.TempTransactionLogFilename);
195         }
196
197         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
198             ↪ useLog: true))
199         {
200             var links = scope.Links;
201             var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;

```

```

196         using (var transaction = transactionsLayer.BeginTransaction())
197         {
198             l2 = links.Update(l2, l1);
199
200             links.Delete(l2);
201
202             ExceptionThrower();
203
204             transaction.Commit();
205         }
206
207         Global.Trash = links.Count();
208     }
209 }
210 catch
211 {
212     Assert.False(lastScope == null);
213
214     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
        ↪ Scope.TempTransactionLogFilename);
215
216     lastScope.DeleteFiles();
217 }
218 }
219
220 [Fact]
221 public static void TransactionCommit()
222 {
223     var itself = _constants.Itself;
224
225     var tempDatabaseFilename = Path.GetTempFileName();
226     var tempTransactionLogFilename = Path.GetTempFileName();
227
228     // Commit
229     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
        ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
        ↪ tempTransactionLogFilename))
230     using (var links = new UInt64Links(memoryAdapter))
231     {
232         using (var transaction = memoryAdapter.BeginTransaction())
233         {
234             var l1 = links.CreateAndUpdate(itself, itself);
235             var l2 = links.CreateAndUpdate(itself, itself);
236
237             Global.Trash = links.Update(l2, l2, l1, l2);
238
239             links.Delete(l1);
240
241             transaction.Commit();
242         }
243
244         Global.Trash = links.Count();
245     }
246
247     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
        ↪ sactionLogFilename);
248 }
249
250 [Fact]
251 public static void TransactionDamage()
252 {
253     var itself = _constants.Itself;
254
255     var tempDatabaseFilename = Path.GetTempFileName();
256     var tempTransactionLogFilename = Path.GetTempFileName();
257
258     // Commit
259     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
        ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
        ↪ tempTransactionLogFilename))
260     using (var links = new UInt64Links(memoryAdapter))
261     {
262         using (var transaction = memoryAdapter.BeginTransaction())
263         {
264             var l1 = links.CreateAndUpdate(itself, itself);
265             var l2 = links.CreateAndUpdate(itself, itself);
266
267             Global.Trash = links.Update(l2, l2, l1, l2);
268

```

```

269         links.Delete(l1);
270
271         transaction.Commit();
272     }
273
274     Global.Trash = links.Count();
275 }
276
277 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
    ↪ sactionLogFilename);
278
279 // Damage database
280
281 FileHelpers.WriteFirst(tempTransactionLogFilename, new
    ↪ UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
282
283 // Try load damaged database
284 try
285 {
286     // TODO: Fix
287     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
        ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
        ↪ 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
        ↪ yet.");
296 }
297
298 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
    ↪ sactionLogFilename);
299
300 File.Delete(tempDatabaseFilename);
301 File.Delete(tempTransactionLogFilename);
302 }
303
304 [Fact]
305 public static void Bug1Test()
306 {
307     var tempDatabaseFilename = Path.GetTempFileName();
308     var tempTransactionLogFilename = Path.GetTempFileName();
309
310     var itself = _constants.Itself;
311
312     // User Code Error (Autoreverted), some data saved
313     try
314     {
315         ulong l1;
316         ulong l2;
317
318         using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
            ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
            ↪ tempTransactionLogFilename))
319         using (var links = new UInt64Links(memoryAdapter))
320         {
321             l1 = links.CreateAndUpdate(itself, itself);
322             l2 = links.CreateAndUpdate(itself, itself);
323
324             l2 = links.Update(l2, l2, l1, l2);
325
326             links.CreateAndUpdate(l2, itself);
327             links.CreateAndUpdate(l2, itself);
328         }
329
330         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp_
            ↪ TransactionLogFilename);
331
332         using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
            ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
            ↪ tempTransactionLogFilename))
333         using (var links = new UInt64Links(memoryAdapter))
334         {
335             using (var transaction = memoryAdapter.BeginTransaction())

```

```

336         {
337             l2 = links.Update(l2, l1);
338             links.Delete(l2);
339             ExceptionThrower();
340             transaction.Commit();
341         }
342         Global.Trash = links.Count();
343     }
344 }
345 catch
346 {
347     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp,
348     ↪ TransactionLogFilename);
349 }
350 File.Delete(tempDatabaseFilename);
351 File.Delete(tempTransactionLogFilename);
352 }
353
354 private static void ExceptionThrower() => throw new InvalidOperationException();
355
356 [Fact]
357 public static void PathsTest()
358 {
359     var source = _constants.SourcePart;
360     var target = _constants.TargetPart;
361
362     using (var scope = new TempLinksTestScope())
363     {
364         var links = scope.Links;
365         var l1 = links.CreatePoint();
366         var l2 = links.CreatePoint();
367
368         var r1 = links.GetByKeys(l1, source, target, source);
369         var r2 = links.CheckPathExistence(l2, l2, l2, l2);
370     }
371 }
372
373 [Fact]
374 public static void RecursiveStringFormattingTest()
375 {
376     using (var scope = new TempLinksTestScope(useSequences: true))
377     {
378         var links = scope.Links;
379         var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.
380
381         var a = links.CreatePoint();
382         var b = links.CreatePoint();
383         var c = links.CreatePoint();
384
385         var ab = links.CreateAndUpdate(a, b);
386         var cb = links.CreateAndUpdate(c, b);
387         var ac = links.CreateAndUpdate(a, c);
388
389         a = links.Update(a, c, b);
390         b = links.Update(b, a, c);
391         c = links.Update(c, a, b);
392
393         Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
394         Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
395         Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
396
397         Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
398         ↪ "(5:(4:5 (6:5 4)) 6)");
399         Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
400         ↪ "(6:(5:(4:5 6) 6) 4)");
401         Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
402         ↪ "(4:(5:4 (6:5 4)) 6)");
403
404         // TODO: Think how to build balanced syntax tree while formatting structure (eg.
405         ↪ "(4:(5:4 6) (6:5 4))" instead of "(4:(5:4 (6:5 4)) 6)"
406
407         Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
408         ↪ "{5}{5}{4}{6}");

```

```

408         Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
409             ↳ "{5}{6}{6}{4}");
410         Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
411             ↳ "{4}{5}{4}{6}");
412     }
413 }
414
415 private static void DefaultFormatter(StringBuilder sb, ulong link)
416 {
417     sb.Append(link.ToString());
418 }
419
420 #endregion
421
422 #region Performance
423
424 /*
425 public static void RunAllPerformanceTests()
426 {
427     try
428     {
429         links.TestLinksInSteps();
430     }
431     catch (Exception ex)
432     {
433         ex.WriteToConsole();
434     }
435
436     return;
437
438     try
439     {
440         //ThreadPool.SetMaxThreads(2, 2);
441         ↳ результат // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
442         // Также это дополнительно помогает в отладке
443         // Увеличивает вероятность попадания информации в кэши
444         for (var i = 0; i < 10; i++)
445         {
446             //0 - 10 ГБ
447             //Каждые 100 МБ срез цифр
448
449             //links.TestGetSourceFunction();
450             //links.TestGetSourceFunctionInParallel();
451             //links.TestGetTargetFunction();
452             //links.TestGetTargetFunctionInParallel();
453             links.Create64BillionLinks();
454
455             links.TestRandomSearchFixed();
456             //links.Create64BillionLinksInParallel();
457             links.TestEachFunction();
458             //links.TestForeach();
459             //links.TestParallelForeach();
460         }
461
462         links.TestDeletionOfAllLinks();
463     }
464     catch (Exception ex)
465     {
466         ex.WriteToConsole();
467     }
468 }*/
469
470 /*
471 public static void TestLinksInSteps()
472 {
473     const long gibibyte = 1024 * 1024 * 1024;
474     const long mebibyte = 1024 * 1024;
475
476     var totalLinksToCreate = gibibyte /
477     ↳ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
478     var linksStep = 102 * mebibyte /
479     ↳ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
480
481     var creationMeasurements = new List<TimeSpan>();
482     var searchMeasurements = new List<TimeSpan>();
483     var deletionMeasurements = new List<TimeSpan>();

```

```

482
483     GetBaseRandomLoopOverhead(linksStep);
484     GetBaseRandomLoopOverhead(linksStep);
485
486     var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
487
488     ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
489
490     var loops = totalLinksToCreate / linksStep;
491
492     for (int i = 0; i < loops; i++)
493     {
494         creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
495         searchMeasurements.Add(Measure(() => links.RunRandomSearches(linksStep)));
496
497         Console.WriteLine("\rC + S {0}/{1}", i + 1, loops);
498     }
499
500     ConsoleHelpers.Debug();
501
502     for (int i = 0; i < loops; i++)
503     {
504         deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
505
506         Console.WriteLine("\rD {0}/{1}", i + 1, loops);
507     }
508
509     ConsoleHelpers.Debug();
510
511     ConsoleHelpers.Debug("C S D");
512
513     for (int i = 0; i < loops; i++)
514     {
515         ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
↵ searchMeasurements[i], deletionMeasurements[i]);
516     }
517
518     ConsoleHelpers.Debug("C S D (no overhead)");
519
520     for (int i = 0; i < loops; i++)
521     {
522         ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] - stepLoopOverhead,
↵ searchMeasurements[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
523     }
524
525     ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
↵ links.Total);
526 }
527
528 private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
↵ amountToCreate)
529 {
530     for (long i = 0; i < amountToCreate; i++)
531         links.Create(0, 0);
532 }
533
534 private static TimeSpan GetBaseRandomLoopOverhead(long loops)
535 {
536     return Measure(() =>
537     {
538         ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
539         ulong result = 0;
540         for (long i = 0; i < loops; i++)
541         {
542             var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
543             var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
544
545             result += maxValue + source + target;
546         }
547         Global.Trash = result;
548     });
549 }
550
551 /*
552 [Fact(Skip = "performance test")]
553 public static void GetSourceTest()
554 {
555     using (var scope = new TempLinksTestScope())
556     {

```

```

557     var links = scope.Links;
558     ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
        ↪ Iterations);
559
560     ulong counter = 0;
561
562     //var firstLink = links.First();
563     // Создаём одну связь, из которой будет производить считывание
564     var firstLink = links.Create();
565
566     var sw = Stopwatch.StartNew();
567
568     // Тестируем саму функцию
569     for (ulong i = 0; i < Iterations; i++)
570     {
571         counter += links.GetSource(firstLink);
572     }
573
574     var elapsedTime = sw.Elapsed;
575
576     var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
577
578     // Удаляем связь, из которой производилось считывание
579     links.Delete(firstLink);
580
581     ConsoleHelpers.Debug(
582         "{0} Iterations of GetSource function done in {1} ({2} Iterations per
        ↪ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
583     }
584 }
585
586 [Fact(Skip = "performance test")]
587 public static void GetSourceInParallel()
588 {
589     using (var scope = new TempLinksTestScope())
590     {
591         var links = scope.Links;
592         ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
        ↪ parallel.", Iterations);
593
594         long counter = 0;
595
596         //var firstLink = links.First();
597         var firstLink = links.Create();
598
599         var sw = Stopwatch.StartNew();
600
601         // Тестируем саму функцию
602         Parallel.For(0, Iterations, x =>
603         {
604             Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
605             //Interlocked.Increment(ref counter);
606         });
607
608         var elapsedTime = sw.Elapsed;
609
610         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
611
612         links.Delete(firstLink);
613
614         ConsoleHelpers.Debug(
615             "{0} Iterations of GetSource function done in {1} ({2} Iterations per
        ↪ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
616     }
617 }
618
619 [Fact(Skip = "performance test")]
620 public static void TestGetTarget()
621 {
622     using (var scope = new TempLinksTestScope())
623     {
624         var links = scope.Links;
625         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
        ↪ Iterations);
626
627         ulong counter = 0;
628
629
630

```

```

631         //var firstLink = links.First();
632         var firstLink = links.Create();
633
634         var sw = Stopwatch.StartNew();
635
636         for (ulong i = 0; i < Iterations; i++)
637         {
638             counter += links.GetTarget(firstLink);
639         }
640
641         var elapsedTime = sw.Elapsed;
642
643         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
644
645         links.Delete(firstLink);
646
647         ConsoleHelpers.Debug(
648             "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
        ↪ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
649     }
650 }
651
652 [Fact(Skip = "performance test")]
653 public static void TestGetTargetInParallel()
654 {
655     using (var scope = new TempLinksTestScope())
656     {
657         var links = scope.Links;
658         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
        ↪ parallel.", Iterations);
659
660         long counter = 0;
661
662         //var firstLink = links.First();
663         var firstLink = links.Create();
664
665         var sw = Stopwatch.StartNew();
666
667         Parallel.For(0, Iterations, x =>
668         {
669             Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
670             //Interlocked.Increment(ref counter);
671         });
672
673         var elapsedTime = sw.Elapsed;
674
675         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
676
677         links.Delete(firstLink);
678
679         ConsoleHelpers.Debug(
680             "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
        ↪ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
681     }
682 }
683
684 // TODO: Заполнить базу данных перед тестом
685 /*
686 [Fact]
687 public void TestRandomSearchFixed()
688 {
689     var tempFilename = Path.GetTempFileName();
690
691     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
        ↪ DefaultLinksSizeStep))
692     {
693         long iterations = 64 * 1024 * 1024 /
        ↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
694
695         ulong counter = 0;
696         var maxLink = links.Total;
697
698         ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);
699
700         var sw = Stopwatch.StartNew();
701
702         for (var i = iterations; i > 0; i--)
703         {
704
705

```



```

706         var source =
↪ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
707         var target =
↪ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
708
709         counter += links.Search(source, target);
710     }
711
712     var elapsedTime = sw.Elapsed;
713
714     var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
715
716     ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
↪ Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
↪ counter);
717 }
718
719     File.Delete(tempFilename);
720 }*/
721
722 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
723 public static void TestRandomSearchAll()
724 {
725     using (var scope = new TempLinksTestScope())
726     {
727         var links = scope.Links;
728         ulong counter = 0;
729
730         var maxLink = links.Count();
731
732         var iterations = links.Count();
733
734         ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
↪ links.Count());
735
736         var sw = Stopwatch.StartNew();
737
738         for (var i = iterations; i > 0; i--)
739         {
740             var linksAddressRange = new
↪ Range<ulong>(_constants.PossibleInnerReferencesRange.Minimum, maxLink);
741
742             var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
743             var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
744
745             counter += links.SearchOrDefault(source, target);
746         }
747
748         var elapsedTime = sw.Elapsed;
749
750         var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
751
752         ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
↪ Iterations per second), c: {3}",
↪ iterations, elapsedTime, (long)iterationsPerSecond, counter);
753     }
754 }
755
756 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
757 public static void TestEach()
758 {
759     using (var scope = new TempLinksTestScope())
760     {
761         var links = scope.Links;
762
763         var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
764
765         ConsoleHelpers.Debug("Testing Each function.");
766
767         var sw = Stopwatch.StartNew();
768
769         links.Each(counter.IncrementAndReturnTrue);
770
771         var elapsedTime = sw.Elapsed;
772
773         var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
774
775         ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
↪ links per second)",
↪ counter, elapsedTime, (long)linksPerSecond);
776
777     }

```

```

778     }
779 }
780
781 /*
782 [Fact]
783 public static void TestForeach()
784 {
785     var tempFilename = Path.GetTempFileName();
786
787     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪ DefaultLinksSizeStep))
788     {
789         ulong counter = 0;
790
791         ConsoleHelpers.Debug("Testing foreach through links.");
792
793         var sw = Stopwatch.StartNew();
794
795         //foreach (var link in links)
796         //{
797             counter++;
798         //}
799
800         var elapsedTime = sw.Elapsed;
801
802         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
803
804         ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
↪ links per second)", counter, elapsedTime, (long)linksPerSecond);
805     }
806
807     File.Delete(tempFilename);
808 }
809 */
810
811 /*
812 [Fact]
813 public static void TestParallelForeach()
814 {
815     var tempFilename = Path.GetTempFileName();
816
817     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪ DefaultLinksSizeStep))
818     {
819         long counter = 0;
820
821         ConsoleHelpers.Debug("Testing parallel foreach through links.");
822
823         var sw = Stopwatch.StartNew();
824
825         //Parallel.ForEach((IEnumerable<ulong>)links, x =>
826         //{
827             //    Interlocked.Increment(ref counter);
828         //});
829
830         var elapsedTime = sw.Elapsed;
831
832         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
833
834         ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
↪ {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
835     }
836
837     File.Delete(tempFilename);
838 }
839 */
840
841 [Fact(Skip = "performance test")]
842 public static void Create64BillionLinks()
843 {
844     using (var scope = new TempLinksTestScope())
845     {
846         var links = scope.Links;
847         var linksBeforeTest = links.Count();
848
849         long linksToCreate = 64 * 1024 * 1024 /
↪ UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
851
852         ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);

```

```

853
854     var elapsedTime = Performance.Measure(() =>
855     {
856         for (long i = 0; i < linksToCreate; i++)
857         {
858             links.Create();
859         }
860     });
861
862     var linksCreated = links.Count() - linksBeforeTest;
863     var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
864
865     ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
866
867     ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
868         ↪ linksCreated, elapsedTime,
869         (long)linksPerSecond);
870 }
871
872 [Fact(Skip = "performance test")]
873 public static void Create64BillionLinksInParallel()
874 {
875     using (var scope = new TempLinksTestScope())
876     {
877         var links = scope.Links;
878         var linksBeforeTest = links.Count();
879
880         var sw = Stopwatch.StartNew();
881
882         long linksToCreate = 64 * 1024 * 1024 /
883             ↪ UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
884
885         ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);
886
887         Parallel.For(0, linksToCreate, x => links.Create());
888
889         var elapsedTime = sw.Elapsed;
890
891         var linksCreated = links.Count() - linksBeforeTest;
892         var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
893
894         ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
895             ↪ linksCreated, elapsedTime,
896             (long)linksPerSecond);
897     }
898 }
899
900 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
901 public static void TestDeletionOfAllLinks()
902 {
903     using (var scope = new TempLinksTestScope())
904     {
905         var links = scope.Links;
906         var linksBeforeTest = links.Count();
907
908         ConsoleHelpers.Debug("Deleting all links");
909
910         var elapsedTime = Performance.Measure(links.DeleteAll);
911
912         var linksDeleted = linksBeforeTest - links.Count();
913         var linksPerSecond = linksDeleted / elapsedTime.TotalSeconds;
914
915         ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
916             ↪ linksDeleted, elapsedTime,
917             (long)linksPerSecond);
918     }
919 }
920
921 #endregion
922 }

```

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

```

1 using Xunit;
2 using Platform.Random;
3 using Platform.Data.Doublets.Numbers.Unary;
4
5 namespace Platform.Data.Doublets.Tests
6 {

```

```

7     public static class UnaryNumberConvertersTests
8     {
9         [Fact]
10        public static void ConvertersTest()
11        {
12            using (var scope = new TempLinksTestScope())
13            {
14                const int N = 10;
15                var links = scope.Links;
16                var meaningRoot = links.CreatePoint();
17                var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
18                var powerOf2ToUnaryNumberConverter = new
19                ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, one);
20                var toUnaryNumberConverter = new AddressToUnaryNumberConverter<ulong>(links,
21                ↪ powerOf2ToUnaryNumberConverter);
22                var random = new System.Random(0);
23                ulong[] numbers = new ulong[N];
24                ulong[] unaryNumbers = new ulong[N];
25                for (int i = 0; i < N; i++)
26                {
27                    numbers[i] = random.NextUInt64();
28                    unaryNumbers[i] = toUnaryNumberConverter.Convert(numbers[i]);
29                }
30                var fromUnaryNumberConverterUsingOrOperation = new
31                ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links,
32                ↪ powerOf2ToUnaryNumberConverter);
33                var fromUnaryNumberConverterUsingAddOperation = new
34                ↪ UnaryNumberToAddressAddOperationConverter<ulong>(links, one);
35                for (int i = 0; i < N; i++)
36                {
37                    Assert.Equal(numbers[i],
38                    ↪ fromUnaryNumberConverterUsingOrOperation.Convert(unaryNumbers[i]));
39                    Assert.Equal(numbers[i],
40                    ↪ fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));
41                }
42            }
43        }
44    }
45 }

```

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

```

1  using Xunit;
2  using Platform.Interfaces;
3  using Platform.Memory;
4  using Platform.Reflection;
5  using Platform.Scopes;
6  using Platform.Data.Doublets.Incrementers;
7  using Platform.Data.Doublets.Numbers.Raw;
8  using Platform.Data.Doublets.Numbers.Unary;
9  using Platform.Data.Doublets.PropertyOperators;
10 using Platform.Data.Doublets.ResizableDirectMemory;
11 using Platform.Data.Doublets.Sequences.Converters;
12 using Platform.Data.Doublets.Sequences.Indexes;
13 using Platform.Data.Doublets.Sequences.Walkers;
14 using Platform.Data.Doublets.Unicode;
15
16 namespace Platform.Data.Doublets.Tests
17 {
18     public static class UnicodeConvertersTests
19     {
20         [Fact]
21         public static void CharAndUnaryNumberUnicodeSymbolConvertersTest()
22         {
23             using (var scope = new TempLinksTestScope())
24             {
25                 var links = scope.Links;
26                 var meaningRoot = links.CreatePoint();
27                 var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
28                 var powerOf2ToUnaryNumberConverter = new
29                 ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, one);
30                 var addressToUnaryNumberConverter = new
31                 ↪ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
32                 var unaryNumberToAddressConverter = new
33                 ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links,
34                 ↪ powerOf2ToUnaryNumberConverter);
35                 TestCharAndUnicodeSymbolConverters(links, meaningRoot,
36                 ↪ addressToUnaryNumberConverter, unaryNumberToAddressConverter);
37             }
38         }
39     }
40 }

```

```

34 [Fact]
35 public static void CharAndRawNumberUnicodeSymbolConvertersTest()
36 {
37     using (var scope = new Scope<Types<HeapResizableDirectMemory,
38         ↳ ResizableDirectMemoryLinks<ulong>>>())
39     {
40         var links = scope.Use<ILinks<ulong>>();
41         var meaningRoot = links.CreatePoint();
42         var addressToRawNumberConverter = new AddressToRawNumberConverter<ulong>();
43         var rawNumberToAddressConverter = new RawNumberToAddressConverter<ulong>();
44         TestCharAndUnicodeSymbolConverters(links, meaningRoot,
45             ↳ addressToRawNumberConverter, rawNumberToAddressConverter);
46     }
47 }
48 private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
49     ↳ meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
50     ↳ numberToAddressConverter)
51 {
52     var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
53     var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links,
54         ↳ addressToNumberConverter, unicodeSymbolMarker);
55     var originalCharacter = 'H';
56     var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);
57     var unicodeSymbolCriterionMatcher = new UnicodeSymbolCriterionMatcher<ulong>(links,
58         ↳ unicodeSymbolMarker);
59     var unicodeSymbolToCharConverter = new UnicodeSymbolToCharConverter<ulong>(links,
60         ↳ numberToAddressConverter, unicodeSymbolCriterionMatcher);
61     var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
62     Assert.Equal(originalCharacter, resultingCharacter);
63 }
64 [Fact]
65 public static void StringAndUnicodeSequenceConvertersTest()
66 {
67     using (var scope = new TempLinksTestScope())
68     {
69         var links = scope.Links;
70         var itself = links.Constants.Itself;
71
72         var meaningRoot = links.CreatePoint();
73         var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
74         var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
75         var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
76         var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
77         var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);
78
79         var powerOf2ToUnaryNumberConverter = new
80             ↳ PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
81         var addressToUnaryNumberConverter = new
82             ↳ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
83         var charToUnicodeSymbolConverter = new
84             ↳ CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
85             ↳ unicodeSymbolMarker);
86
87         var unaryNumberToAddressConverter = new
88             ↳ UnaryNumberToAddressOrOperationConverter<ulong>(links,
89             ↳ powerOf2ToUnaryNumberConverter);
90         var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
91         var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
92             ↳ frequencyMarker, unaryOne, unaryNumberIncrementer);
93         var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
94             ↳ frequencyPropertyMarker, frequencyMarker);
95         var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
96             ↳ frequencyPropertyOperator, frequencyIncrementer);
97         var linkToItsFrequencyNumberConverter = new
98             ↳ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
99             ↳ unaryNumberToAddressConverter);
100        var sequenceToItsLocalElementLevelsConverter = new
101            ↳ SequenceToItsLocalElementLevelsConverter<ulong>(links,
102            ↳ linkToItsFrequencyNumberConverter);
103        var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
104            ↳ sequenceToItsLocalElementLevelsConverter);

```

```

89     var stringToUnicodeSequenceConverter = new
    ↪     StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
    ↪     index, optimalVariantConverter, unicodeSequenceMarker);
90
91     var originalString = "Hello";
92
93     var unicodeSequenceLink =
    ↪     stringToUnicodeSequenceConverter.Convert(originalString);
94
95     var unicodeSymbolCriterionMatcher = new
    ↪     UnicodeSymbolCriterionMatcher<ulong>(links, unicodeSymbolMarker);
96     var unicodeSymbolToCharConverter = new
    ↪     UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
    ↪     unicodeSymbolCriterionMatcher);
97
98     var unicodeSequenceCriterionMatcher = new
    ↪     UnicodeSequenceCriterionMatcher<ulong>(links, unicodeSequenceMarker);
99
100    var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
    ↪     unicodeSymbolCriterionMatcher.IsMatched);
101
102    var unicodeSequenceToStringConverter = new
    ↪     UnicodeSequenceToStringConverter<ulong>(links,
    ↪     unicodeSequenceCriterionMatcher, sequenceWalker,
    ↪     unicodeSymbolToCharConverter);
103
104    var resultingString =
    ↪     unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
105
106    Assert.Equal(originalString, resultingString);
107    }
108    }
109    }
110 }

```

Index

./Platform.Data.Doublets.Tests/ComparisonTests.cs, 142
./Platform.Data.Doublets.Tests/EqualityTests.cs, 143
./Platform.Data.Doublets.Tests/GenericLinksTests.cs, 144
./Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs, 145
./Platform.Data.Doublets.Tests/ReadSequenceTests.cs, 147
./Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs, 147
./Platform.Data.Doublets.Tests/ScopeTests.cs, 148
./Platform.Data.Doublets.Tests/SequencesTests.cs, 149
./Platform.Data.Doublets.Tests/TempLinksTestScope.cs, 164
./Platform.Data.Doublets.Tests/TestExtensions.cs, 164
./Platform.Data.Doublets.Tests/UInt64LinksTests.cs, 167
./Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs, 179
./Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs, 180
./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/ILinksListMethods.cs, 34
./Platform.Data.Doublets/ResizableDirectMemory/ILinksTreeMethods.cs, 34
./Platform.Data.Doublets/ResizableDirectMemory/LinksAVLBalancedTreeMethodsBase.cs, 34
./Platform.Data.Doublets/ResizableDirectMemory/LinksHeader.cs, 38
./Platform.Data.Doublets/ResizableDirectMemory/LinksSourcesAVLBalancedTreeMethods.cs, 38
./Platform.Data.Doublets/ResizableDirectMemory/LinksTargetsAVLBalancedTreeMethods.cs, 39
./Platform.Data.Doublets/ResizableDirectMemory/RawLink.cs, 41
./Platform.Data.Doublets/ResizableDirectMemory/ResizableDirectMemoryLinks.cs, 41
./Platform.Data.Doublets/ResizableDirectMemory/UInt64LinksAVLBalancedTreeMethodsBase.cs, 48
./Platform.Data.Doublets/ResizableDirectMemory/UInt64LinksHeader.cs, 51
./Platform.Data.Doublets/ResizableDirectMemory/UInt64LinksSourcesAVLBalancedTreeMethods.cs, 52
./Platform.Data.Doublets/ResizableDirectMemory/UInt64LinksTargetsAVLBalancedTreeMethods.cs, 53
./Platform.Data.Doublets/ResizableDirectMemory/UInt64RawLink.cs, 54
./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.cs, 54
./Platform.Data.Doublets/ResizableDirectMemory/UInt64UnusedLinksListMethods.cs, 61
./Platform.Data.Doublets/ResizableDirectMemory/UnusedLinksListMethods.cs, 62
./Platform.Data.Doublets/Sequences/ArrayExtensions.cs, 62
./Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.cs, 63
./Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs, 63

./Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs, 66
./Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs, 67
./Platform.Data.Doublets/Sequences/Converters/SequenceToltsLocalElementLevelsConverter.cs, 68
./Platform.Data.Doublets/Sequences/CreteriaMatchers/DefaultSequenceElementCriterionMatcher.cs, 69
./Platform.Data.Doublets/Sequences/CreteriaMatchers/MarkedSequenceCriterionMatcher.cs, 69
./Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs, 69
./Platform.Data.Doublets/Sequences/DuplicateSegmentsCounter.cs, 70
./Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs, 70
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 72
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs, 74
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToltsFrequencyValueConverter.cs, 75
./Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 75
./Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 75
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 76
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 76
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 77
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 77
./Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 78
./Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 78
./Platform.Data.Doublets/Sequences/HeightProviders/ISequenceHeightProvider.cs, 79
./Platform.Data.Doublets/Sequences/IListExtensions.cs, 79
./Platform.Data.Doublets/Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs, 79
./Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs, 80
./Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.cs, 81
./Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.cs, 81
./Platform.Data.Doublets/Sequences/Indexes/SynchronizedSequenceIndex.cs, 82
./Platform.Data.Doublets/Sequences/Indexes/Unindex.cs, 82
./Platform.Data.Doublets/Sequences/ListFiller.cs, 82
./Platform.Data.Doublets/Sequences/Sequences.Experiments.cs, 93
./Platform.Data.Doublets/Sequences/Sequences.cs, 83
./Platform.Data.Doublets/Sequences/SequencesExtensions.cs, 119
./Platform.Data.Doublets/Sequences/SequencesOptions.cs, 120
./Platform.Data.Doublets/Sequences/SetFiller.cs, 121
./Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.cs, 122
./Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs, 122
./Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs, 123
./Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs, 124
./Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs, 125
./Platform.Data.Doublets/Stacks/Stack.cs, 126
./Platform.Data.Doublets/Stacks/StackExtensions.cs, 126
./Platform.Data.Doublets/SynchronizedLinks.cs, 127
./Platform.Data.Doublets/UInt64Link.cs, 127
./Platform.Data.Doublets/UInt64LinkExtensions.cs, 130
./Platform.Data.Doublets/UInt64LinksExtensions.cs, 130
./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs, 132
./Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs, 137
./Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs, 138
./Platform.Data.Doublets/Unicode/UnicodeMap.cs, 138
./Platform.Data.Doublets/Unicode/UnicodeSequenceCriterionMatcher.cs, 141
./Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs, 141
./Platform.Data.Doublets/Unicode/UnicodeSymbolCriterionMatcher.cs, 141
./Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs, 142