

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],
        ↳ substitution[constants.TargetPart]);
20         return Links.Update(restrictions, substitution);
21     }
22 }
23 }

```

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

```

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

```

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

```

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

```

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

```

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

```

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

```

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

```

```

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

```

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

```

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

```

```

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

```



```

135     //{
136     //     Memory.FreeLink(restriction[Constants.IndexPart]);
137     //}
138     //else if (restriction.EqualTo(substitution)) // Read or ("repeat" the state) // Each
139     //{
140     //     // No need to collect links to list
141     //     // Skip == Continue
142     //     // No need to check substitutedHandler
143     //     if (!Memory.Each(link => !Equals(matchedHandler(Memory.GetLinkValue(link)),
144     ↪ Constants.Break), restriction))
145     //         return Constants.Break;
146     //}
147     //else // Update
148     //{
149     //     //List<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             {

```

```

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

```

./Platform.Data.Doubles/Douplet.cs

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

./Platform.Data.Doubles/Hybrid.cs

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

```

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

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

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

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

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

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

```

```

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

```

./Platform.Data.Doublets/ILinks.cs

```

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

```

./Platform.Data.Doublets/ILinksExtensions.cs

```

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

```

```

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

```

```

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

```



```

177     /// </summary>
178     /// <param name="links">Хранилище связей.</param>
179     /// <param name="link">Связь представленная списком, состоящим из её адреса и
    ↳ содержимого.</param>
180     /// <returns>Индекс начальной связи для указанной связи.</returns>
181     [MethodImpl(MethodImplOptions.AggressiveInlining)]
182     public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
    ↳ link[links.Constants.IndexPart];
183
184     /// <summary>
185     /// Возвращает индекс начальной (Source) связи для указанной связи.
186     /// </summary>
187     /// <param name="links">Хранилище связей.</param>
188     /// <param name="link">Индекс связи.</param>
189     /// <returns>Индекс начальной связи для указанной связи.</returns>
190     [MethodImpl(MethodImplOptions.AggressiveInlining)]
191     public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink link) =>
    ↳ links.GetLink(link)[links.Constants.SourcePart];
192
193     /// <summary>
194     /// Возвращает индекс начальной (Source) связи для указанной связи.
195     /// </summary>
196     /// <param name="links">Хранилище связей.</param>
197     /// <param name="link">Связь представленная списком, состоящим из её адреса и
    ↳ содержимого.</param>
198     /// <returns>Индекс начальной связи для указанной связи.</returns>
199     [MethodImpl(MethodImplOptions.AggressiveInlining)]
200     public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
    ↳ link[links.Constants.SourcePart];
201
202     /// <summary>
203     /// Возвращает индекс конечной (Target) связи для указанной связи.
204     /// </summary>
205     /// <param name="links">Хранилище связей.</param>
206     /// <param name="link">Индекс связи.</param>
207     /// <returns>Индекс конечной связи для указанной связи.</returns>
208     [MethodImpl(MethodImplOptions.AggressiveInlining)]
209     public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
    ↳ links.GetLink(link)[links.Constants.TargetPart];
210
211     /// <summary>
212     /// Возвращает индекс конечной (Target) связи для указанной связи.
213     /// </summary>
214     /// <param name="links">Хранилище связей.</param>
215     /// <param name="link">Связь представленная списком, состоящим из её адреса и
    ↳ содержимого.</param>
216     /// <returns>Индекс конечной связи для указанной связи.</returns>
217     [MethodImpl(MethodImplOptions.AggressiveInlining)]
218     public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
    ↳ link[links.Constants.TargetPart];
219
220     /// <summary>
221     /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
    ↳ (handler) для каждой подходящей связи.
222     /// </summary>
223     /// <param name="links">Хранилище связей.</param>
224     /// <param name="handler">Обработчик каждой подходящей связи.</param>
225     /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
    ↳ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
    ↳ Any - отсутствие ограничения, 1..∞ конкретный адрес связи.</param>
226     /// <returns>True, в случае если проход по связям не был прерван и False в обратном
    ↳ случае.</returns>
227     [MethodImpl(MethodImplOptions.AggressiveInlining)]
228     public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
    ↳ handler, params TLink[] restrictions)
229     => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
    ↳ links.Constants.Continue);
230
231     /// <summary>
232     /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
    ↳ (handler) для каждой подходящей связи.
233     /// </summary>
234     /// <param name="links">Хранилище связей.</param>
235     /// <param name="source">Значение, определяющее соответствующие шаблону связи.
    ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
    ↳ Constants.Any - любое начало, 1..∞ конкретное начало)</param>

```

```

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

```

```

295 public static bool Exists<TLink>(this ILinks<TLink> links, TLink source, TLink target)
296     => Comparer<TLink>.Default.Compare(links.Count(links.Constants.Any, source, target),
297     => default) > 0;
298
299 #region Ensure
300 // TODO: May be move to EnsureExtensions or make it both there and here
301 [MethodImpl(MethodImplOptions.AggressiveInlining)]
302 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
303     => reference, string argumentName)
304 {
305     if (links.Constants.IsInnerReference(reference) && !links.Exists(reference))
306     {
307         throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
308     }
309 }
310 [MethodImpl(MethodImplOptions.AggressiveInlining)]
311 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
312     => IList<TLink> restrictions, string argumentName)
313 {
314     for (int i = 0; i < restrictions.Count; i++)
315     {
316         links.EnsureInnerReferenceExists(restrictions[i], argumentName);
317     }
318 }
319 [MethodImpl(MethodImplOptions.AggressiveInlining)]
320 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
321     => restrictions)
322 {
323     for (int i = 0; i < restrictions.Count; i++)
324     {
325         links.EnsureLinkIsAnyOrExists(restrictions[i], nameof(restrictions));
326     }
327 }
328 [MethodImpl(MethodImplOptions.AggressiveInlining)]
329 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
330     => string argumentName)
331 {
332     var equalityComparer = EqualityComparer<TLink>.Default;
333     if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
334     {
335         throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
336     }
337 }
338 [MethodImpl(MethodImplOptions.AggressiveInlining)]
339 public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links, TLink
340     => link, string argumentName)
341 {
342     var equalityComparer = EqualityComparer<TLink>.Default;
343     if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
344     {
345         throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
346     }
347 }
348
349 /// <param name="links">Хранилище связей.</param>
350 [MethodImpl(MethodImplOptions.AggressiveInlining)]
351 public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink source,
352     => TLink target)
353 {
354     if (links.Exists(source, target))
355     {
356         throw new LinkWithSameValueAlreadyExistsException();
357     }
358 }
359
360 /// <param name="links">Хранилище связей.</param>
361 [MethodImpl(MethodImplOptions.AggressiveInlining)]
362 public static void EnsureNoUsages<TLink>(this ILinks<TLink> links, TLink link)
363 {
364     if (links.HasUsages(link))
365     {
366         throw new ArgumentLinkHasDependenciesException<TLink>(link);
367     }
368 }

```

```

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.. $\infty$  конкретный адрес другой
    ↳ связи.</param>
484 /// <returns>Индекс обновлённой связи.</returns>
485 [MethodImpl(MethodImplOptions.AggressiveInlining)]
486 public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
487 {
488     if (restrictions.Length == 2)
489     {
490         return links.MergeAndDelete(restrictions[0], restrictions[1]);
491     }
492     if (restrictions.Length == 4)
493     {
494         return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
            ↳ restrictions[2], restrictions[3]);
495     }
496     else
497     {

```

```

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

```

```

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

```

```

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

```



```

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

```

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

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Incrementers
7  {
8      public class FrequencyIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11         ↪ EqualityComparer<TLink>.Default;
12
13         private readonly TLink _frequencyMarker;
14         private readonly TLink _unaryOne;
15         private readonly IIncrementer<TLink> _unaryNumberIncrementer;
16
17         public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
18         ↪ IIncrementer<TLink> unaryNumberIncrementer)
19         : base(links)
20         {
21             _frequencyMarker = frequencyMarker;
22             _unaryOne = unaryOne;
23             _unaryNumberIncrementer = unaryNumberIncrementer;
24         }
25
26         public TLink Increment(TLink frequency)
27         {
28             if (_equalityComparer.Equals(frequency, default))
29             {
30                 return Links.GetOrCreate(_unaryOne, _frequencyMarker);
31             }
32             var source = Links.GetSource(frequency);
33             var incrementedSource = _unaryNumberIncrementer.Increment(source);
34             return Links.GetOrCreate(incrementedSource, _frequencyMarker);
35         }
36     }
37 }

```

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

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Incrementers
7  {
8      public class UnaryNumberIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11         ↪ EqualityComparer<TLink>.Default;
12
13         private readonly TLink _unaryOne;

```

```

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

```

./Platform.Data.Doublets/ISynchronizedLinks.cs

```

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

```

./Platform.Data.Doublets/Link.cs

```

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

```

```

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

```

```

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

```

./Platform.Data.Doublets/LinkExtensions.cs

```

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

```

./Platform.Data.Doublets/LinksOperatorBase.cs

```

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

```

```

9     }
10 }

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

```

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

```

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

```

1  using System.Collections.Generic;
2  using Platform.Exceptions;
3  using Platform.Interfaces;
4  using Platform.Ranges;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Numbers.Unary
9  {
10     public class PowerOf2ToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
11         ↪ IConverter<int, TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ↪ EqualityComparer<TLink>.Default;
15
16         private readonly TLink[] _unaryNumberPowersOf2;
17
18         public PowerOf2ToUnaryNumberConverter(ILinks<TLink> links, TLink one) : base(links)
19         {
20             _unaryNumberPowersOf2 = new TLink[64];
21             _unaryNumberPowersOf2[0] = one;
22         }
23
24         public TLink Convert(int power)
25         {
26             Ensure.Always.ArgumentInRange(power, new Range<int>(0, _unaryNumberPowersOf2.Length
27                 ↪ - 1), nameof(power));
28             if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
29             {
30                 return _unaryNumberPowersOf2[power];
31             }
32             var previousPowerOf2 = Convert(power - 1);
33             var powerOf2 = Links.GetOrCreate(previousPowerOf2, previousPowerOf2);
34             _unaryNumberPowersOf2[power] = powerOf2;
35             return powerOf2;
36         }
37     }
38 }

```

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

```
1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Interfaces;
4 using Platform.Numbers;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Numbers.Unary
9 {
10     public class UnaryNumberToAddressAddOperationConverter<TLink> : LinksOperatorBase<TLink>,
11         ⇨ IConverter<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ⇨ EqualityComparer<TLink>.Default;
15
16         private Dictionary<TLink, TLink> _unaryToUInt64;
17         private readonly TLink _unaryOne;
18
19         public UnaryNumberToAddressAddOperationConverter(ILinks<TLink> links, TLink unaryOne)
20             : base(links)
21         {
22             _unaryOne = unaryOne;
23             InitUnaryToUInt64();
24         }
25
26         private void InitUnaryToUInt64()
27         {
28             var one = Integer<TLink>.One;
29             _unaryToUInt64 = new Dictionary<TLink, TLink>
30             {
31                 { _unaryOne, one }
32             };
33             var unary = _unaryOne;
34             var number = one;
35             for (var i = 1; i < 64; i++)
36             {
37                 unary = Links.GetOrCreate(unary, unary);
38                 number = Double(number);
39                 _unaryToUInt64.Add(unary, number);
40             }
41         }
42
43         public TLink Convert(TLink unaryNumber)
44         {
45             if (_equalityComparer.Equals(unaryNumber, default))
46             {
47                 return default;
48             }
49             if (_equalityComparer.Equals(unaryNumber, _unaryOne))
50             {
51                 return Integer<TLink>.One;
52             }
53             var source = Links.GetSource(unaryNumber);
54             var target = Links.GetTarget(unaryNumber);
55             if (_equalityComparer.Equals(source, target))
56             {
57                 return _unaryToUInt64[unaryNumber];
58             }
59             else
60             {
61                 var result = _unaryToUInt64[source];
62                 TLink lastValue;
63                 while (!_unaryToUInt64.TryGetValue(target, out lastValue))
64                 {
65                     source = Links.GetSource(target);
66                     result = Arithmetic<TLink>.Add(result, _unaryToUInt64[source]);
67                     target = Links.GetTarget(target);
68                 }
69                 result = Arithmetic<TLink>.Add(result, lastValue);
70                 return result;
71             }
72         }
73
74         [MethodImpl(MethodImplOptions.AggressiveInlining)]
75         private static TLink Double(TLink number) => (Integer<TLink>)((Integer<TLink>)number *
76             ⇨ 2UL);
77     }
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)]
76     }
```



```

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             [MethodImpl(MethodImplOptions.AggressiveInlining)]
68             protected override ulong Decrement(ulong value) => --value;
69
70             [MethodImpl(MethodImplOptions.AggressiveInlining)]
71             protected override bool IsLessThanZero(ulong value) => value < OUL;
72
73             [MethodImpl(MethodImplOptions.AggressiveInlining)]
74             protected override bool IsLessThan(ulong first, ulong second) => first < second;
75
76             [MethodImpl(MethodImplOptions.AggressiveInlining)]
77             protected override bool IsGreaterOrEqualThanZero(ulong value) => value > OUL;
78
79             [MethodImpl(MethodImplOptions.AggressiveInlining)]
80             protected override bool IsGreaterOrEqualThan(ulong first, ulong second) => first >= second;
81
82             [MethodImpl(MethodImplOptions.AggressiveInlining)]
83             protected override bool IsEquals(ulong first, ulong second) => first == second;
84
85             [MethodImpl(MethodImplOptions.AggressiveInlining)]
86             protected override bool EqualToZero(ulong value) => value == OUL;
87
88             [MethodImpl(MethodImplOptions.AggressiveInlining)]
89             protected override ulong GetZero() => OUL;
90
91             [MethodImpl(MethodImplOptions.AggressiveInlining)]
92             protected override void Dispose(bool manual, bool wasDisposed)
93             {
94                 if (!wasDisposed)
95                 {
96                     SetPointers(null);
97                     _memory.DisposeIfPossible();
98                 }
99             }
100         }
101     }
102 }

```



```

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 FirstIsToTheLeftOfSecond(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 }

```

./Platform.Data.Doublets/ResizableDirectMemory/UInt64LinksTargetsAVLBalancedTreeMethods.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 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)]

```

```

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             }
304         }
305     }
306 }

```

```

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     ///     A 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.IsMatch(_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

```

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(doupletIndex);
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(ICollection<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(ICollection<TLink> sequence)
41          {
42              var links = _links.Unsync;
43              return _links.SyncRoot.ExecuteReadOperation(() =>
44              {
45                  var indexed = true;
46                  var i = sequence.Count;
47                  while (--i >= 1 && (indexed =
48                     ↳ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
49                     ↳ sequence[i]), default))) { }
50                  return indexed;
51              });
52          }
53      }
54 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences
7  {
8      public class ListFiller<TElement, TReturnConstant>
9      {
10          protected readonly List<TElement> _list;
11          protected readonly TReturnConstant _returnConstant;
12
13          public ListFiller(List<TElement> list, TReturnConstant returnConstant)
14          {
15              _list = list;
16              _returnConstant = returnConstant;
17          }
18

```

```

19     public ListFiller(List<TElement> list) : this(list, default) { }
20
21     [MethodImpl(MethodImplOptions.AggressiveInlining)]
22     public void Add(TElement element) => _list.Add(element);
23
24     [MethodImpl(MethodImplOptions.AggressiveInlining)]
25     public bool AddAndReturnTrue(TElement element)
26     {
27         _list.Add(element);
28         return true;
29     }
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     public bool AddFirstAndReturnTrue(ICollection<TElement> collection)
33     {
34         _list.Add(collection[0]);
35         return true;
36     }
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     public TReturnConstant AddAndReturnConstant(TElement element)
40     {
41         _list.Add(element);
42         return _returnConstant;
43     }
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     public TReturnConstant AddFirstAndReturnConstant(ICollection<TElement> collection)
47     {
48         _list.Add(collection[0]);
49         return _returnConstant;
50     }
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     public TReturnConstant AddAllValuesAndReturnConstant(ICollection<TElement> collection)
54     {
55         for (int i = 1; i < collection.Count; i++)
56         {
57             _list.Add(collection[i]);
58         }
59         return _returnConstant;
60     }
61 }
62 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections;
6  using Platform.Collections.Lists;
7  using Platform.Threading.Synchronization;
8  using Platform.Singletons;
9  using LinkIndex = System.UInt64;
10 using Platform.Data.Doublets.Sequences.Walkers;
11 using Platform.Collections.Stacks;
12 using Platform.Collections.Arrays;
13
14 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16 namespace Platform.Data.Doublets.Sequences
17 {
18     /// <summary>
19     /// Представляет коллекцию последовательностей связей.
20     /// </summary>
21     /// <remarks>
22     /// Обязательно реализовать атомарность каждого публичного метода.
23     ///
24     /// TODO:
25     ///
26     /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
27     /// через естественную группировку по unicode типам, все whitespace вместе, все символы
28     /// вместе, все числа вместе и т.п.
29     /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
30     /// графа)
31     ///
32     /// х*у - найти все связи между, в последовательностях любой формы, если не стоит
33     /// ограничитель на то, что является последовательностью, а что нет,

```

```

31  /// то находятся любые структуры связей, которые содержат эти элементы именно в таком
32  ↪ порядке.
33  ///
34  /// Рост последовательности слева и справа.
35  /// Поиск со звёздочкой.
36  /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
37  /// так же проблема может быть решена при реализации дистанционных триггеров.
38  /// Нужны ли уникальные указатели вообще?
39  /// Что если обращение к информации будет происходить через содержимое всегда?
40  ///
41  /// Писать тесты.
42  ///
43  /// Можно убрать зависимость от конкретной реализации Links,
44  ↪ на зависимость от абстрактного элемента, который может быть представлен несколькими
45  ↪ способами.
46  ///
47  /// Можно ли как-то сделать один общий интерфейс
48  ///
49  /// Блокчейн и/или гит для распределённой записи транзакций.
50  ///
51  </remarks>
52  public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
53  ↪ (после завершения реализации Sequences)
54  {
55  /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
56  ↪ связей.</summary>
57  public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;
58
59  public SequencesOptions<LinkIndex> Options { get; }
60  public SynchronizedLinks<LinkIndex> Links { get; }
61  private readonly ISynchronization _sync;
62
63  public LinksConstants<LinkIndex> Constants { get; }
64
65  public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
66  {
67  Links = links;
68  _sync = links.SyncRoot;
69  Options = options;
70  Options.ValidateOptions();
71  Options.InitOptions(Links);
72  Constants = Default<LinksConstants<LinkIndex>>.Instance;
73  }
74
75  public Sequences(SynchronizedLinks<LinkIndex> links)
76  : this(links, new SequencesOptions<LinkIndex>())
77  {
78  }
79
80  public bool IsSequence(LinkIndex sequence)
81  {
82  return _sync.ExecuteReadOperation(() =>
83  {
84  if (Options.UseSequenceMarker)
85  {
86  return Options.MarkedSequenceMatcher.IsMatched(sequence);
87  }
88  return !Links.Unsync.IsPartialPoint(sequence);
89  });
90  }
91
92  [MethodImpl(MethodImplOptions.AggressiveInlining)]
93  private LinkIndex GetSequenceByElements(LinkIndex sequence)
94  {
95  if (Options.UseSequenceMarker)
96  {
97  return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
98  }
99  return sequence;
100  }
101
102  private LinkIndex GetSequenceElements(LinkIndex sequence)
103  {
104  if (Options.UseSequenceMarker)
105  {
106  var linkContents = new UInt64Link(Links.GetLink(sequence));
107  if (linkContents.Source == Options.SequenceMarkerLink)

```

```

106         {
107             return linkContents.Target;
108         }
109         if (linkContents.Target == Options.SequenceMarkerLink)
110         {
111             return linkContents.Source;
112         }
113     }
114     return sequence;
115 }
116
117 #region Count
118
119 public LinkIndex Count(IList<LinkIndex> restrictions)
120 {
121     if (restrictions.IsNullOrEmpty())
122     {
123         return Links.Count(Constants.Any, Options.SequenceMarkerLink, Constants.Any);
124     }
125     if (restrictions.Count == 1) // Первая связь это адрес
126     {
127         var sequenceIndex = restrictions[0];
128         if (sequenceIndex == Constants.Null)
129         {
130             return 0;
131         }
132         if (sequenceIndex == Constants.Any)
133         {
134             return Count(null);
135         }
136         if (Options.UseSequenceMarker)
137         {
138             return Links.Count(Constants.Any, Options.SequenceMarkerLink, sequenceIndex);
139         }
140         return Links.Exists(sequenceIndex) ? 1UL : 0;
141     }
142     throw new NotImplementedException();
143 }
144
145 private LinkIndex CountUsages(params LinkIndex[] restrictions)
146 {
147     if (restrictions.Length == 0)
148     {
149         return 0;
150     }
151     if (restrictions.Length == 1) // Первая связь это адрес
152     {
153         if (restrictions[0] == Constants.Null)
154         {
155             return 0;
156         }
157         if (Options.UseSequenceMarker)
158         {
159             var elementsLink = GetSequenceElements(restrictions[0]);
160             var sequenceLink = GetSequenceByElements(elementsLink);
161             if (sequenceLink != Constants.Null)
162             {
163                 return Links.Count(sequenceLink) + Links.Count(elementsLink) - 1;
164             }
165             return Links.Count(elementsLink);
166         }
167         return Links.Count(restrictions[0]);
168     }
169     throw new NotImplementedException();
170 }
171
172 #endregion
173
174 #region Create
175
176 public LinkIndex Create(IList<LinkIndex> restrictions)
177 {
178     return _sync.ExecuteWriteOperation(() =>
179     {
180         if (restrictions.IsNullOrEmpty())
181         {
182             return Constants.Null;
183         }
184         Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));

```

```

185         return CreateCore(restrictions);
186     });
187 }
188
189 private LinkIndex CreateCore(ICollection<LinkIndex> restrictions)
190 {
191     LinkIndex[] sequence = restrictions.ExtractValues();
192     if (Options.UseIndex)
193     {
194         Options.Index.Add(sequence);
195     }
196     var sequenceRoot = default(LinkIndex);
197     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
198     {
199         var matches = Each(restrictions);
200         if (matches.Count > 0)
201         {
202             sequenceRoot = matches[0];
203         }
204     }
205     else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
206     {
207         return CompactCore(sequence);
208     }
209     if (sequenceRoot == default)
210     {
211         sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
212     }
213     if (Options.UseSequenceMarker)
214     {
215         Links.Unsync.CreateAndUpdate(Options.SequenceMarkerLink, sequenceRoot);
216     }
217     return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
218 }
219
220 #endregion
221
222 #region Each
223
224 public List<LinkIndex> Each(ICollection<LinkIndex> sequence)
225 {
226     var results = new List<LinkIndex>();
227     var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
228     Each(filler.AddFirstAndReturnConstant, sequence);
229     return results;
230 }
231
232 public LinkIndex Each(Func<ICollection<LinkIndex>, LinkIndex> handler, ICollection<LinkIndex>
    ↳ restrictions)
233 {
234     return _sync.ExecuteReadOperation(() =>
235     {
236         if (restrictions.IsNullOrEmpty())
237         {
238             return Constants.Continue;
239         }
240         Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
241         if (restrictions.Count == 1)
242         {
243             var link = restrictions[0];
244             var any = Constants.Any;
245             if (link == any)
246             {
247                 if (Options.UseSequenceMarker)
248                 {
249                     return Links.Unsync.Each(handler, new Link<LinkIndex>(any,
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     });
264 }

```

```

259     }
260     else if (restrictions.Count == 2)
261     {
262         throw new NotImplementedException();
263     }
264     else if (restrictions.Count == 3)
265     {
266         return Links.Unsync.Each(handler, restrictions);
267     }
268     else
269     {
270         var sequence = restrictions.ExtractValues();
271         if (Options.UseIndex && !Options.Index.MightContain(sequence))
272         {
273             return Constants.Break;
274         }
275         return EachCore(handler, sequence);
276     }
277     });
278 }
279
280 private LinkIndex EachCore(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
    ↪ values)
281 {
282     var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
283     // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
    ↪ Id.
284     Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
    ↪ (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
    ↪ matcher.HandleFullMatched;
285     //if (sequence.Length >= 2)
286     if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
287     {
288         return Constants.Break;
289     }
290     var last = values.Count - 2;
291     for (var i = 1; i < last; i++)
292     {
293         if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
    ↪ Constants.Continue)
294         {
295             return Constants.Break;
296         }
297     }
298     if (values.Count >= 3)
299     {
300         if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
    ↪ != Constants.Continue)
301         {
302             return Constants.Break;
303         }
304     }
305     return Constants.Continue;
306 }
307
308 private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↪ left, LinkIndex right)
309 {
310     return Links.Unsync.Each(doublet =>
311     {
312         var doubletIndex = doublet[Constants.IndexPart];
313         if (StepRight(handler, doubletIndex, right) != Constants.Continue)
314         {
315             return Constants.Break;
316         }
317         if (left != doubletIndex)
318         {
319             return PartialStepRight(handler, doubletIndex, right);
320         }
321         return Constants.Continue;
322     }, new Link<LinkIndex>(Constants.Any, Constants.Any, left));
323 }
324
325 private LinkIndex StepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
    ↪ LinkIndex right) => Links.Unsync.Each(rightStep => TryStepRightUp(handler, right,
    ↪ rightStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, left,
    ↪ Constants.Any));
326

```

```

327 private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↳ right, LinkIndex stepFrom)
328 {
329     var upStep = stepFrom;
330     var firstSource = Links.Unsync.GetTarget(upStep);
331     while (firstSource != right && firstSource != upStep)
332     {
333         upStep = firstSource;
334         firstSource = Links.Unsync.GetSource(upStep);
335     }
336     if (firstSource == right)
337     {
338         return handler(new LinkAddress<LinkIndex>(stepFrom));
339     }
340     return Constants.Continue;
341 }
342
343 private LinkIndex StepLeft(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
    ↳ LinkIndex right) => Links.Unsync.Each(leftStep => TryStepLeftUp(handler, left,
    ↳ leftStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, Constants.Any,
    ↳ right));
344
345 private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↳ left, LinkIndex stepFrom)
346 {
347     var upStep = stepFrom;
348     var firstTarget = Links.Unsync.GetSource(upStep);
349     while (firstTarget != left && firstTarget != upStep)
350     {
351         upStep = firstTarget;
352         firstTarget = Links.Unsync.GetTarget(upStep);
353     }
354     if (firstTarget == left)
355     {
356         return handler(new LinkAddress<LinkIndex>(stepFrom));
357     }
358     return Constants.Continue;
359 }
360
361 #endregion
362
363 #region Update
364
365 public LinkIndex Update(IList<LinkIndex> restrictions, IList<LinkIndex> substitution)
366 {
367     var sequence = restrictions.ExtractValues();
368     var newSequence = substitution.ExtractValues();
369
370     if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
371     {
372         return Constants.Null;
373     }
374     if (sequence.IsNullOrEmpty())
375     {
376         return Create(substitution);
377     }
378     if (newSequence.IsNullOrEmpty())
379     {
380         Delete(restrictions);
381         return Constants.Null;
382     }
383     return _sync.ExecuteWriteOperation(() =>
384     {
385         Links.EnsureEachLinkIsAnyOrExists(sequence);
386         Links.EnsureEachLinkExists(newSequence);
387         return UpdateCore(sequence, newSequence);
388     });
389 }
390
391 private LinkIndex UpdateCore(LinkIndex[] sequence, LinkIndex[] newSequence)
392 {
393     LinkIndex bestVariant;
394     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
    ↳ !sequence.EqualTo(newSequence))
395     {
396         bestVariant = CompactCore(newSequence);
397     }
398     else
399     {

```



```

400     bestVariant = CreateCore(newSequence);
401 }
402 // TODO: Check all options only ones before loop execution
403 // Возможно нужно две версии Each, возвращающий фактические последовательности и с
404   ↳ маркером,
405 // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
406   ↳ можно получить имея только фактические последовательности.
407 foreach (var variant in Each(sequence))
408 {
409     if (variant != bestVariant)
410     {
411         UpdateOneCore(variant, bestVariant);
412     }
413 }
414 return bestVariant;
415 }
416
417 private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
418 {
419     if (Options.UseGarbageCollection)
420     {
421         var sequenceElements = GetSequenceElements(sequence);
422         var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
423         var sequenceLink = GetSequenceByElements(sequenceElements);
424         var newSequenceElements = GetSequenceElements(newSequence);
425         var newSequenceLink = GetSequenceByElements(newSequenceElements);
426         if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
427         {
428             if (sequenceLink != Constants.Null)
429             {
430                 Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
431             }
432             Links.Unsync.MergeUsages(sequenceElements, newSequenceElements);
433         }
434         ClearGarbage(sequenceElementsContents.Source);
435         ClearGarbage(sequenceElementsContents.Target);
436     }
437     else
438     {
439         if (Options.UseSequenceMarker)
440         {
441             var sequenceElements = GetSequenceElements(sequence);
442             var sequenceLink = GetSequenceByElements(sequenceElements);
443             var newSequenceElements = GetSequenceElements(newSequence);
444             var newSequenceLink = GetSequenceByElements(newSequenceElements);
445             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
446             {
447                 if (sequenceLink != Constants.Null)
448                 {
449                     Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
450                 }
451                 Links.Unsync.MergeUsages(sequenceElements, newSequenceElements);
452             }
453         }
454         else
455         {
456             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
457             {
458                 Links.Unsync.MergeUsages(sequence, newSequence);
459             }
460         }
461     }
462 }
463
464 #endregion
465
466 #region Delete
467
468 public void Delete(IList<LinkIndex> restrictions)
469 {
470     _sync.ExecuteWriteOperation(() =>
471     {
472         var sequence = restrictions.ExtractValues();
473         // TODO: Check all options only ones before loop execution
474         foreach (var linkToDelete in Each(sequence))
475         {
476             DeleteOneCore(linkToDelete);
477         }
478     })
479 }

```

```

476     });
477 }
478
479 private void DeleteOneCore(LinkIndex link)
480 {
481     if (Options.UseGarbageCollection)
482     {
483         var sequenceElements = GetSequenceElements(link);
484         var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
485         var sequenceLink = GetSequenceByElements(sequenceElements);
486         if (Options.UseCascadeDelete || CountUsages(link) == 0)
487         {
488             if (sequenceLink != Constants.Null)
489             {
490                 Links.Unsync.Delete(sequenceLink);
491             }
492             Links.Unsync.Delete(link);
493         }
494         ClearGarbage(sequenceElementsContents.Source);
495         ClearGarbage(sequenceElementsContents.Target);
496     }
497     else
498     {
499         if (Options.UseSequenceMarker)
500         {
501             var sequenceElements = GetSequenceElements(link);
502             var sequenceLink = GetSequenceByElements(sequenceElements);
503             if (Options.UseCascadeDelete || CountUsages(link) == 0)
504             {
505                 if (sequenceLink != Constants.Null)
506                 {
507                     Links.Unsync.Delete(sequenceLink);
508                 }
509                 Links.Unsync.Delete(link);
510             }
511         }
512         else
513         {
514             if (Options.UseCascadeDelete || CountUsages(link) == 0)
515             {
516                 Links.Unsync.Delete(link);
517             }
518         }
519     }
520 }
521
522 #endregion
523
524 #region Compactification
525
526 /// <remarks>
527 /// bestVariant можно выбирать по максимальному числу использований,
528 /// но балансированный позволяет гарантировать уникальность (если есть возможность,
529 /// гарантировать его использование в других местах).
530 ///
531 /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
532 /// </remarks>
533 public LinkIndex Compact(params LinkIndex[] sequence)
534 {
535     return _sync.ExecuteWriteOperation(() =>
536     {
537         if (sequence.IsNullOrEmpty())
538         {
539             return Constants.Null;
540         }
541         Links.EnsureEachLinkExists(sequence);
542         return CompactCore(sequence);
543     });
544 }
545
546 [MethodImpl(MethodImplOptions.AggressiveInlining)]
547 private LinkIndex CompactCore(params LinkIndex[] sequence) => UpdateCore(sequence,
548     ↪ sequence);
549
550 #endregion
551
552 #region Garbage Collection
553
554 /// <remarks>

```

```

554 /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
555   → определить извне или в унаследованном классе
556 /// </remarks>
557 [MethodImpl(MethodImplOptions.AggressiveInlining)]
558 private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
559   → !Links.Unsync.IsPartialPoint(link) && Links.Count(link) == 0;
560
561 private void ClearGarbage(LinkIndex link)
562 {
563     if (IsGarbage(link))
564     {
565         var contents = new UInt64Link(Links.GetLink(link));
566         Links.Unsync.Delete(link);
567         ClearGarbage(contents.Source);
568         ClearGarbage(contents.Target);
569     }
570 }
571
572 #endregion
573
574 #region Walkers
575
576 public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
577 {
578     return _sync.ExecuteReadOperation(() =>
579     {
580         var links = Links.Unsync;
581         foreach (var part in Options.Walker.Walk(sequence))
582         {
583             if (!handler(part))
584             {
585                 return false;
586             }
587         }
588         return true;
589     });
590 }
591
592 public class Matcher : RightSequenceWalker<LinkIndex>
593 {
594     private readonly Sequences _sequences;
595     private readonly IList<LinkIndex> _patternSequence;
596     private readonly HashSet<LinkIndex> _linksInSequence;
597     private readonly HashSet<LinkIndex> _results;
598     private readonly Func<IList<LinkIndex>, LinkIndex> _stopableHandler;
599     private readonly HashSet<LinkIndex> _readAsElements;
600     private int _filterPosition;
601
602     public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
603   → HashSet<LinkIndex> results, Func<IList<LinkIndex>, LinkIndex> stopableHandler,
604   → HashSet<LinkIndex> readAsElements = null)
605         : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
606     {
607         _sequences = sequences;
608         _patternSequence = patternSequence;
609         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
610   → Links.Constants.Any && x != ZeroOrMany));
611         _results = results;
612         _stopableHandler = stopableHandler;
613         _readAsElements = readAsElements;
614     }
615
616     protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
617   → (_readAsElements != null && _readAsElements.Contains(link)) ||
618   → _linksInSequence.Contains(link);
619
620     public bool FullMatch(LinkIndex sequenceToMatch)
621     {
622         _filterPosition = 0;
623         foreach (var part in Walk(sequenceToMatch))
624         {
625             if (!FullMatchCore(part))
626             {
627                 break;
628             }
629         }
630         return _filterPosition == _patternSequence.Count;
631     }
632
633     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))
693     {
694         return false; // Нашлось
695     }
696     if (_filterPosition >= 0)
697     {
698         if (element == _patternSequence[_filterPosition + 1])
699         {
700             _filterPosition++;
701         }
702         else
703         {
704             _filterPosition = -1;
705         }
706     }
707 }

```

```

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
85     public List<ulong> CreateAllVariants1(params ulong[] sequence)
86     {
87         return _sync.ExecuteWriteOperation(() =>
88         {
89             if (sequence.IsNullOrEmpty())
90             {
91                 return new List<ulong>();
92             }
93             Links.Unsync.EnsureEachLinkExists(sequence);
94             if (sequence.Length == 1)
95             {
96                 return new List<ulong> { sequence[0] };
97             }
98             var results = new
99                 ↪ List<ulong>((int)Platform.Numbers.Math.Catalan(sequence.Length));
100             return CreateAllVariants1Core(sequence, results);

```

```

96     });
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;
161         for (var li = 0; li < innerSequenceLength; li++)
162         {
163             var left = sequence[li];
164             var right = sequence[li + 1];
165             if (left == 0 && right == 0)
166             {
167                 continue;
168             }
169             var linkIndex = li;
170             ulong[] innerSequence = null;
171             Links.Unsync.Each(doublet =>
172             {
173                 if (innerSequence == null)
174                 {

```

```

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
237         {
238             Links.Each(Constants.Any, Constants.Any, handler);
239         }
240     }
241     else if (sequence.Length == 2)
242     {
243         //_links.Each(sequence[0], sequence[1], handler);
244         //  o_|      x_o ...
245         // x_|      |___|
246         Links.Each(sequence[1], Constants.Any, doublet =>
247         {
248             var match = Links.SearchOrDefault(sequence[0], doublet);
249             if (match != Constants.Null)
250             {

```



```

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

```

```

Links.Unsync.Each(Constants.Any, right, leftStep =>
{
    TryStepLeftUp(handler, left, leftStep);
    return true;
});
}

private void TryStepLeftUp(Action<IList<LinkIndex>> handler, ulong left, ulong stepFrom)
{
    var upStep = stepFrom;
    var firstTarget = Links.Unsync.GetSource(upStep);
    while (firstTarget != left && firstTarget != upStep)
    {
        upStep = firstTarget;
        firstTarget = Links.Unsync.GetTarget(upStep);
    }
    if (firstTarget == left)
    {
        handler(new LinkAddress<LinkIndex>(stepFrom));
    }
}

private bool StartsWith(ulong sequence, ulong link)
{
    var upStep = sequence;
    var firstSource = Links.Unsync.GetSource(upStep);
    while (firstSource != link && firstSource != upStep)
    {
        upStep = firstSource;
        firstSource = Links.Unsync.GetSource(upStep);
    }
    return firstSource == link;
}

private bool EndsWith(ulong sequence, ulong link)
{
    var upStep = sequence;
    var lastTarget = Links.Unsync.GetTarget(upStep);
    while (lastTarget != link && lastTarget != upStep)
    {
        upStep = lastTarget;
        lastTarget = Links.Unsync.GetTarget(upStep);
    }
    return lastTarget == link;
}

public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
{
    return _sync.ExecuteReadOperation(() =>
    {
        var results = new List<ulong>();
        if (sequence.Length > 0)
        {
            Links.EnsureEachLinkExists(sequence);
            var firstElement = sequence[0];
            if (sequence.Length == 1)
            {
                results.Add(firstElement);
                return results;
            }
            if (sequence.Length == 2)
            {
                var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
                if (doublet != Constants.Null)
                {
                    results.Add(doublet);
                }
                return results;
            }
            var linksInSequence = new HashSet<ulong>(sequence);
            void handler(IList<LinkIndex> result)
            {
                var resultIndex = result[Links.Constants.IndexPart];
                var filterPosition = 0;
                StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
                    ↪ Links.Unsync.GetTarget,
                    x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
                    ↪ x =>
                {

```

```

        if (filterPosition == sequence.Length)
        {
            filterPosition = -2; // Длиннее чем нужно
            return false;
        }
        if (x != sequence[filterPosition])
        {
            filterPosition = -1;
            return false; // Начинается иначе
        }
        filterPosition++;

        return true;
    });
    if (filterPosition == sequence.Length)
    {
        results.Add(resultIndex);
    }
}
if (sequence.Length >= 2)
{
    StepRight(handler, sequence[0], sequence[1]);
}
var last = sequence.Length - 2;
for (var i = 1; i < last; i++)
{
    PartialStepRight(handler, sequence[i], sequence[i + 1]);
}
if (sequence.Length >= 3)
{
    StepLeft(handler, sequence[sequence.Length - 2],
        ↪ sequence[sequence.Length - 1]);
}
}
return results;
});
}

public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
{
    return _sync.ExecuteReadOperation(() =>
    {
        var results = new HashSet<ulong>();
        if (sequence.Length > 0)
        {
            Links.EnsureEachLinkExists(sequence);
            var firstElement = sequence[0];
            if (sequence.Length == 1)
            {
                results.Add(firstElement);
                return results;
            }
            if (sequence.Length == 2)
            {
                var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
                if (doublet != Constants.Null)
                {
                    results.Add(doublet);
                }
                return results;
            }
            var matcher = new Matcher(this, sequence, results, null);
            if (sequence.Length >= 2)
            {
                StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
            }
            var last = sequence.Length - 2;
            for (var i = 1; i < last; i++)
            {
                PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
                    ↪ sequence[i + 1]);
            }
            if (sequence.Length >= 3)
            {
                StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
                    ↪ sequence[sequence.Length - 1]);
            }
        }
    });
}

```

```

480         return results;
481     });
482 }
483
484 public const int MaxSequenceFormatSize = 200;
485
486 public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
487     => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
488
489 public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
490     elementToString, bool insertComma, params LinkIndex[] knownElements) =>
491     Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
492         elementToString, insertComma, knownElements));
493
494 private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
495     Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
496     LinkIndex[] knownElements)
497 {
498     var linksInSequence = new HashSet<ulong>(knownElements);
499     //var entered = new HashSet<ulong>();
500     var sb = new StringBuilder();
501     sb.Append('{');
502     if (links.Exists(sequenceLink))
503     {
504         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
505             x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
506             entered.AddAndReturnVoid, x => { }, entered.DoNotContains
507             {
508                 if (insertComma && sb.Length > 1)
509                 {
510                     sb.Append(',');
511                 }
512                 //if (entered.Contains(element))
513                 //{
514                 //    sb.Append('{');
515                 //    elementToString(sb, element);
516                 //    sb.Append('}');
517                 //}
518                 //else
519                 elementToString(sb, element);
520                 if (sb.Length < MaxSequenceFormatSize)
521                 {
522                     return true;
523                 }
524                 sb.Append(insertComma ? ", ..." : "...");
525                 return false;
526             }
527     });
528     sb.Append('}');
529     return sb.ToString();
530 }
531
532 public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
533     knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
534     knownElements);
535
536 public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
537     LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
538     Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
539     sequenceLink, elementToString, insertComma, knownElements));
540
541 private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
542     Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
543     LinkIndex[] knownElements)
544 {
545     var linksInSequence = new HashSet<ulong>(knownElements);
546     var entered = new HashSet<ulong>();
547     var sb = new StringBuilder();
548     sb.Append('{');
549     if (links.Exists(sequenceLink))
550     {
551         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
552             x => linksInSequence.Contains(x) || links.IsFullPoint(x),
553             entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
554             {
555                 if (insertComma && sb.Length > 1)
556                 {

```

```

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

```

```

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                     return true;
661                 }
662                 return true;
663             }
664             return false;
665         });
666     }
667
668     //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
669     //{
670     //    return Sync.ExecuteReadOperation(() =>
671     //    {
672     //        if (sequence.Length > 0)
673     //        {
674     //            _links.EnsureEachLinkIsAnyOrExists(sequence);
675
676     //            var firstResults = new HashSet<ulong>();
677     //            var lastResults = new HashSet<ulong>();
678
679     //            var first = sequence.First(x => x != LinksConstants.Any);
680     //            var last = sequence.Last(x => x != LinksConstants.Any);
681
682     //            AllUsagesCore(first, firstResults);
683     //            AllUsagesCore(last, lastResults);
684
685     //            firstResults.IntersectWith(lastResults);
686
687     //            //for (var i = 0; i < sequence.Length; i++)
688     //            //    AllUsagesCore(sequence[i], results);
689
690     //            var filteredResults = new HashSet<ulong>();
691     //            var matcher = new Matcher(this, sequence, filteredResults, null);
692     //            matcher.AddAllPartialMatchedToResults(firstResults);
693     //            return filteredResults;
694     //        }
695     //    });
696     //    return new HashSet<ulong>();
697     //}

```

```

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);
761             matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
762 ↪ x)); // OrderBy is a Hack
763             return filteredResults;
764         }
765         return new HashSet<ulong>();
766     });
767 }
768
769 // Does not work
770 //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
771 ↪ params ulong[] sequence)
772 //{
773 //    var visited = new HashSet<ulong>();
774 //    var results = new HashSet<ulong>();

```

```

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

```



```

843         return new List<ulong>();
844     }
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 }
906
907 public HashSet<ulong> AllBottomUsages(ulong link)
908 {
909     return _sync.ExecuteReadOperation(() =>
910     {
911         var visits = new HashSet<ulong>();
912         var usages = new HashSet<ulong>();
913         AllBottomUsagesCore(link, visits, usages);
914         return usages;
915     });
916 }
917
918 private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
919     usages)
920 {

```

```

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 }
978
979 public void CalculateAllUsages2(ulong[] totals)
980 {
981     var calculator = new AllUsagesCalculator2(Links, totals);
982     calculator.Calculate();
983 }
984
985 private class AllUsagesCalculator
986 {
987     private readonly SynchronizedLinks<ulong> _links;
988     private readonly ulong[] _totals;
989
990     public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
991     {
992         _links = links;
993         _totals = totals;
994     }
995 }

```

```

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

```

```

1068         if (isElement(element))
1069         {
1070             if (stack.Count == 0)
1071             {
1072                 break;
1073             }
1074             element = stack.Pop();
1075             var source = getSource(element);
1076             var target = getTarget(element);
1077             // Обработка элемента
1078             if (isElement(target))
1079             {
1080                 visitLeaf(target);
1081             }
1082             if (isElement(source))
1083             {
1084                 visitLeaf(source);
1085             }
1086             element = source;
1087         }
1088         else
1089         {
1090             stack.Push(element);
1091             visitNode(element);
1092             element = getTarget(element);
1093         }
1094     }
1095 }
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>
↪ intersectWith, HashSet<ulong> usages)
1177     {
1178         _links = links;
1179         _intersectWith = intersectWith;
1180         _usages = usages;
1181         _enter = new HashSet<ulong>(); // защита от заикливания
1182     }
1183
1184     public bool Collect(ulong link)
1185     {
1186         if (_enter.Add(link))
1187         {
1188             if (_intersectWith.Contains(link))
1189             {
1190                 _usages.Add(link);
1191             }
1192             _links.Unsync.Each(link, _links.Constants.Any, Collect);
1193             _links.Unsync.Each(_links.Constants.Any, link, Collect);
1194         }
1195         return true;
1196     }
1197 }
1198
1199 private void CloseInnerConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
↪ right)
1200 {
1201     TryStepLeftUp(handler, left, right);
1202     TryStepRightUp(handler, right, left);
1203 }
1204
1205 private void AllCloseConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
↪ right)
1206 {
1207     // Direct
1208     if (left == right)
1209     {
1210         handler(new LinkAddress<LinkIndex>(left));
1211     }
1212     var doublet = Links.Unsync.SearchOrDefault(left, right);
1213     if (doublet != Constants.Null)
1214     {
1215         handler(new LinkAddress<LinkIndex>(doublet));
1216     }
1217     // Inner
1218     CloseInnerConnections(handler, left, right);
1219     // Outer
1220     StepLeft(handler, left, right);
1221     StepRight(handler, left, right);
1222     PartialStepRight(handler, left, right);
1223     PartialStepLeft(handler, left, right);

```

```

1224 }
1225
1226 private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
    ↪ HashSet<ulong> previousMatchings, long startAt)
1227 {
1228     if (startAt >= sequence.Length) // ?
1229     {
1230         return previousMatchings;
1231     }
1232     var secondLinkUsages = new HashSet<ulong>();
1233     AllUsagesCore(sequence[startAt], secondLinkUsages);
1234     secondLinkUsages.Add(sequence[startAt]);
1235     var matchings = new HashSet<ulong>();
1236     var filler = new SetFiller<LinkIndex, LinkIndex>(matchings, Constants.Continue);
1237     //for (var i = 0; i < previousMatchings.Count; i++)
1238     foreach (var secondLinkUsage in secondLinkUsages)
1239     {
1240         foreach (var previousMatching in previousMatchings)
1241         {
1242             //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
1243             ↪ secondLinkUsage);
1244             StepRight(filler.AddFirstAndReturnConstant, previousMatching,
1245             ↪ secondLinkUsage);
1246             TryStepRightUp(filler.AddFirstAndReturnConstant, secondLinkUsage,
1247             ↪ previousMatching);
1248             //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
1249             ↪ sequence[startAt]); // почему-то эта ошибочная запись приводит к
1250             ↪ желаемым результатам.
1251             PartialStepRight(filler.AddFirstAndReturnConstant, previousMatching,
1252             ↪ secondLinkUsage);
1253         }
1254     }
1255     if (matchings.Count == 0)
1256     {
1257         return matchings;
1258     }
1259     return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
1260 }
1261
1262 private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
    ↪ links, params ulong[] sequence)
1263 {
1264     if (sequence == null)
1265     {
1266         return;
1267     }
1268     for (var i = 0; i < sequence.Length; i++)
1269     {
1270         if (sequence[i] != links.Constants.Any && sequence[i] != ZeroOrMany &&
1271         ↪ !links.Exists(sequence[i]))
1272         {
1273             throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
1274             ↪ $"patternSequence[{i}]");
1275         }
1276     }
1277 }
1278
1279 // Pattern Matching -> Key To Triggers
1280 public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1281 {
1282     return _sync.ExecuteReadOperation(() =>
1283     {
1284         patternSequence = Simplify(patternSequence);
1285         if (patternSequence.Length > 0)
1286         {
1287             EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1288             var uniqueSequenceElements = new HashSet<ulong>();
1289             for (var i = 0; i < patternSequence.Length; i++)
1290             {
1291                 if (patternSequence[i] != Constants.Any && patternSequence[i] !=
1292                 ↪ ZeroOrMany)
1293                 {
1294                     uniqueSequenceElements.Add(patternSequence[i]);
1295                 }
1296             }
1297             var results = new HashSet<ulong>();
1298             foreach (var uniqueSequenceElement in uniqueSequenceElements)

```

```

1290         {
1291             AllUsagesCore(uniqueSequenceElement, results);
1292         }
1293         var filteredResults = new HashSet<ulong>();
1294         var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1295         matcher.AddAllPatternMatchedToResults(results);
1296         return filteredResults;
1297     }
1298     return new HashSet<ulong>();
1299 });
1300 }
1301
1302 // Найти все возможные связи между указанным списком связей.
1303 // Находит связи между всеми указанными связями в любом порядке.
1304 // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
1305 //        несколько раз в последовательности)
1306 public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1307 {
1308     return _sync.ExecuteReadOperation(() =>
1309     {
1310         var results = new HashSet<ulong>();
1311         if (linksToConnect.Length > 0)
1312         {
1313             Links.EnsureEachLinkExists(linksToConnect);
1314             AllUsagesCore(linksToConnect[0], results);
1315             for (var i = 1; i < linksToConnect.Length; i++)
1316             {
1317                 var next = new HashSet<ulong>();
1318                 AllUsagesCore(linksToConnect[i], next);
1319                 results.IntersectWith(next);
1320             }
1321             return results;
1322         }
1323     });
1324 }
1325
1326 public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
1327 {
1328     return _sync.ExecuteReadOperation(() =>
1329     {
1330         var results = new HashSet<ulong>();
1331         if (linksToConnect.Length > 0)
1332         {
1333             Links.EnsureEachLinkExists(linksToConnect);
1334             var collector1 = new AllUsagesCollector(Links.Unsync, results);
1335             collector1.Collect(linksToConnect[0]);
1336             var next = new HashSet<ulong>();
1337             for (var i = 1; i < linksToConnect.Length; i++)
1338             {
1339                 var collector = new AllUsagesCollector(Links.Unsync, next);
1340                 collector.Collect(linksToConnect[i]);
1341                 results.IntersectWith(next);
1342                 next.Clear();
1343             }
1344             return results;
1345         }
1346     });
1347 }
1348
1349 public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
1350 {
1351     return _sync.ExecuteReadOperation(() =>
1352     {
1353         var results = new HashSet<ulong>();
1354         if (linksToConnect.Length > 0)
1355         {
1356             Links.EnsureEachLinkExists(linksToConnect);
1357             var collector1 = new AllUsagesCollector(Links, results);
1358             collector1.Collect(linksToConnect[0]);
1359             //AllUsagesCore(linksToConnect[0], results);
1360             for (var i = 1; i < linksToConnect.Length; i++)
1361             {
1362                 var next = new HashSet<ulong>();
1363                 var collector = new AllUsagesIntersectingCollector(Links, results, next);
1364                 collector.Collect(linksToConnect[i]);
1365                 //AllUsagesCore(linksToConnect[i], next);
1366                 //results.IntersectWith(next);
1367                 results = next;

```

```

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             return results.GetSetUInt64Indices();
1393         }
1394     });
1395 }
1396
1397 private static ulong[] Simplify(ulong[] sequence)
1398 {
1399     // Считаем новый размер последовательности
1400     long newLength = 0;
1401     var zeroOrManyStepped = false;
1402     for (var i = 0; i < sequence.Length; i++)
1403     {
1404         if (sequence[i] == ZeroOrMany)
1405         {
1406             if (zeroOrManyStepped)
1407             {
1408                 continue;
1409             }
1410             zeroOrManyStepped = true;
1411         }
1412         else
1413         {
1414             //if (zeroOrManyStepped) Is it efficient?
1415             zeroOrManyStepped = false;
1416             newLength++;
1417         }
1418     }
1419     // Строим новую последовательность
1420     zeroOrManyStepped = false;
1421     var newSequence = new ulong[newLength];
1422     long j = 0;
1423     for (var i = 0; i < sequence.Length; i++)
1424     {
1425         //var current = zeroOrManyStepped;
1426         //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
1427         //if (current && zeroOrManyStepped)
1428         //    continue;
1429         //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1430         //if (zeroOrManyStepped && newZeroOrManyStepped)
1431         //    continue;
1432         //zeroOrManyStepped = newZeroOrManyStepped;
1433         if (sequence[i] == ZeroOrMany)
1434         {
1435             if (zeroOrManyStepped)
1436             {
1437                 continue;
1438             }
1439             zeroOrManyStepped = true;
1440         }
1441         else
1442         {
1443             //if (zeroOrManyStepped) Is it efficient?
1444             zeroOrManyStepped = false;
1445             newSequence[j] = sequence[i];
1446             j++;
1447         }
1448     }
1449     return newSequence;
1450 }

```



```

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,
        ↪ ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
1452     var simplifiedSequence = Simplify(sequence);
1453 }
1454
1455 public List<ulong> GetSimilarSequences() => new List<ulong>();
1456
1457 public void Prediction()
1458 {
1459     //_links
1460     //_sequences
1461 }
1462
1463 #region From Triplets
1464
1465 //public static void DeleteSequence(Link sequence)
1466 //{
1467 //}
1468
1469 public List<ulong> CollectMatchingSequences(ulong[] links)
1470 {
1471     if (links.Length == 1)
1472     {
1473         throw new Exception("Подпоследовательности с одним элементом не
        ↪ поддерживаются.");
1474     }
1475     var leftBound = 0;
1476     var rightBound = links.Length - 1;
1477     var left = links[leftBound++];
1478     var right = links[rightBound--];
1479     var results = new List<ulong>();
1480     CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
1481     return results;
1482 }
1483
1484 private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
    ↪ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1485 {
1486     var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1487     var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1488     if (leftLinkTotalReferers <= rightLinkTotalReferers)
1489     {
1490         var nextLeftLink = middleLinks[leftBound];
1491         var elements = GetRightElements(leftLink, nextLeftLink);
1492         if (leftBound <= rightBound)
1493         {
1494             for (var i = elements.Length - 1; i >= 0; i--)
1495             {
1496                 var element = elements[i];
1497                 if (element != 0)
1498                 {
1499                     CollectMatchingSequences(element, leftBound + 1, middleLinks,
        ↪ rightLink, rightBound, ref results);
1500                 }
1501             }
1502         }
1503         else
1504         {
1505             for (var i = elements.Length - 1; i >= 0; i--)
1506             {
1507                 var element = elements[i];
1508                 if (element != 0)
1509                 {
1510                     results.Add(element);
1511                 }
1512             }
1513         }
1514     }
1515     else
1516     {
1517         var nextRightLink = middleLinks[rightBound];

```

```

1518     var elements = GetLeftElements(rightLink, nextRightLink);
1519     if (leftBound <= rightBound)
1520     {
1521         for (var i = elements.Length - 1; i >= 0; i--)
1522         {
1523             var element = elements[i];
1524             if (element != 0)
1525             {
1526                 CollectMatchingSequences(leftLink, leftBound, middleLinks,
1527                     ↪ elements[i], rightBound - 1, ref results);
1528             }
1529         }
1530     }
1531     else
1532     {
1533         for (var i = elements.Length - 1; i >= 0; i--)
1534         {
1535             var element = elements[i];
1536             if (element != 0)
1537             {
1538                 results.Add(element);
1539             }
1540         }
1541     }
1542 }
1543
1544 public ulong[] GetRightElements(ulong startLink, ulong rightLink)
1545 {
1546     var result = new ulong[5];
1547     TryStepRight(startLink, rightLink, result, 0);
1548     Links.Each(Constants.Any, startLink, couple =>
1549     {
1550         if (couple != startLink)
1551         {
1552             if (TryStepRight(couple, rightLink, result, 2))
1553             {
1554                 return false;
1555             }
1556         }
1557         return true;
1558     });
1559     if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
1560     {
1561         result[4] = startLink;
1562     }
1563     return result;
1564 }
1565
1566 public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
1567 {
1568     var added = 0;
1569     Links.Each(startLink, Constants.Any, couple =>
1570     {
1571         if (couple != startLink)
1572         {
1573             var coupleTarget = Links.GetTarget(couple);
1574             if (coupleTarget == rightLink)
1575             {
1576                 result[offset] = couple;
1577                 if (++added == 2)
1578                 {
1579                     return false;
1580                 }
1581             }
1582             else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker
1583                 ↪ == Net.And &&
1584             {
1585                 result[offset + 1] = couple;
1586                 if (++added == 2)
1587                 {
1588                     return false;
1589                 }
1590             }
1591         }
1592     });
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
1653 public class PatternMatcher : RightSequenceWalker<ulong>
1654 {
1655     private readonly Sequences _sequences;
1656     private readonly ulong[] _patternSequence;
1657     private readonly HashSet<LinkIndex> _linksInSequence;
1658     private readonly HashSet<LinkIndex> _results;
1659
1660     #region Pattern Match
1661
1662     enum PatternBlockType
1663     {
1664         Undefined,
1665         Gap,
1666         Elements
1667     }
1668
1669     struct PatternBlock
1670     {
1671         public PatternBlockType Type;
1672         public long Start;
1673         public long Stop;
1674     }
1675 
```

```

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             else if (patternBlock.Type == PatternBlockType.Elements)
1739             {
1740                 if (_patternSequence[i] == _sequences.Constants.Any)
1741                 {
1742                     pattern.Add(patternBlock);
1743                     patternBlock = new PatternBlock
1744                     {
1745                         Type = PatternBlockType.Gap,
1746                         Start = 1,
1747                         Stop = 1
1748                     };
1749                 }
1750                 else if (_patternSequence[i] == ZeroOrMany)
1751                 {
1752                     pattern.Add(patternBlock);
1753                 }
1754             }
1755         }
1756     }

```

```

1749         patternBlock = new PatternBlock
1750         {
1751             Type = PatternBlockType.Gap,
1752             Start = 0,
1753             Stop = long.MaxValue
1754         };
1755     }
1756     else
1757     {
1758         patternBlock.Stop = i;
1759     }
1760 }
1761 else // patternBlock.Type == PatternBlockType.Gap
1762 {
1763     if (_patternSequence[i] == _sequences.Constants.Any)
1764     {
1765         patternBlock.Start++;
1766         if (patternBlock.Stop < patternBlock.Start)
1767         {
1768             patternBlock.Stop = patternBlock.Start;
1769         }
1770     }
1771     else if (_patternSequence[i] == ZeroOrMany)
1772     {
1773         patternBlock.Stop = long.MaxValue;
1774     }
1775     else
1776     {
1777         pattern.Add(patternBlock);
1778         patternBlock = new PatternBlock
1779         {
1780             Type = PatternBlockType.Elements,
1781             Start = i,
1782             Stop = i
1783         };
1784     }
1785 }
1786 }
1787 if (patternBlock.Type != PatternBlockType.Undefined)
1788 {
1789     pattern.Add(patternBlock);
1790 }
1791 return pattern;
1792 }
1793
1794 // match: search for regexp anywhere in text
1795 //int match(char* regexp, char* text)
1796 //{
1797 //    do
1798 //    {
1799 //    } while (*text++ != '\0');
1800 //    return 0;
1801 //}
1802
1803 // matchhere: search for regexp at beginning of text
1804 //int matchhere(char* regexp, char* text)
1805 //{
1806 //    if (regexp[0] == '\0')
1807 //        return 1;
1808 //    if (regexp[1] == '*')
1809 //        return matchstar(regexp[0], regexp + 2, text);
1810 //    if (regexp[0] == '$' && regexp[1] == '\0')
1811 //        return *text == '\0';
1812 //    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
1813 //        return matchhere(regexp + 1, text + 1);
1814 //    return 0;
1815 //}
1816
1817 // matchstar: search for c*regexp at beginning of text
1818 //int matchstar(int c, char* regexp, char* text)
1819 //{
1820 //    do
1821 //    {
1822 //        /* a * matches zero or more instances */
1823 //        if (matchhere(regexp, text))
1824 //            return 1;
1825 //    } while (*text != '\0' && (*text++ == c || c == '.'));
1826 //    return 0;
1827 //}

```

```

1828 //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out
1829 ↪ long maximumGap)
1830 //{
1831 //    mininumGap = 0;
1832 //    maximumGap = 0;
1833 //    element = 0;
1834 //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
1835 //    {
1836 //        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
1837 //            mininumGap++;
1838 //        else if (_patternSequence[_patternPosition] == ZeroOrMany)
1839 //            maximumGap = long.MaxValue;
1840 //        else
1841 //            break;
1842 //    }
1843 //    if (maximumGap < mininumGap)
1844 //        maximumGap = mininumGap;
1845 //}
1846
1847 private bool PatternMatchCore(LinkIndex element)
1848 {
1849     if (_patternPosition >= _pattern.Count)
1850     {
1851         _patternPosition = -2;
1852         return false;
1853     }
1854     var currentPatternBlock = _pattern[_patternPosition];
1855     if (currentPatternBlock.Type == PatternBlockType.Gap)
1856     {
1857         //var currentMatchingBlockLength = (_sequencePosition -
1858         ↪ _lastMatchedBlockPosition);
1859         if (_sequencePosition < currentPatternBlock.Start)
1860         {
1861             _sequencePosition++;
1862             return true; // Двигаемся дальше
1863         }
1864         // Это последний блок
1865         if (_pattern.Count == _patternPosition + 1)
1866         {
1867             _patternPosition++;
1868             _sequencePosition = 0;
1869             return false; // Полное соответствие
1870         }
1871         else
1872         {
1873             if (_sequencePosition > currentPatternBlock.Stop)
1874             {
1875                 return false; // Соответствие невозможно
1876             }
1877             var nextPatternBlock = _pattern[_patternPosition + 1];
1878             if (_patternSequence[nextPatternBlock.Start] == element)
1879             {
1880                 if (nextPatternBlock.Start < nextPatternBlock.Stop)
1881                 {
1882                     _patternPosition++;
1883                     _sequencePosition = 1;
1884                 }
1885                 else
1886                 {
1887                     _patternPosition += 2;
1888                     _sequencePosition = 0;
1889                 }
1890             }
1891         }
1892     }
1893     else // currentPatternBlock.Type == PatternBlockType.Elements
1894     {
1895         var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1896         if (_patternSequence[patternElementPosition] != element)
1897         {
1898             return false; // Соответствие невозможно
1899         }
1900         if (patternElementPosition == currentPatternBlock.Stop)
1901         {
1902             _patternPosition++;
1903             _sequencePosition = 0;
1904         }
1905         else

```

```

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

```

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

```

1  using System;
2  using System.Collections.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences
7  {
8      public static class SequencesExtensions
9      {
10         public static TLink Create<TLink>(this ILinks<TLink> sequences, IList<TLink[]>
            ↳ groupedSequence)
11         {
12             var finalSequence = new TLink[groupedSequence.Count];
13             for (var i = 0; i < finalSequence.Length; i++)
14             {
15                 var part = groupedSequence[i];
16                 finalSequence[i] = part.Length == 1 ? part[0] :
                    ↳ sequences.Create(part.ConvertToRestrictionsValues());
17             }
18             return sequences.Create(finalSequence.ConvertToRestrictionsValues());

```

```

19     }
20
21     public static IList<TLink> ToList<TLink>(this ILinks<TLink> sequences, TLink sequence)
22     {
23         var list = new List<TLink>();
24         var filler = new ListFiller<TLink, TLink>(list, sequences.Constants.Break);
25         sequences.Each(filler.AddAllValuesAndReturnConstant, new
26             ↪ LinkAddress<TLink>(sequence));
27         return list;
28     }
29 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4  using Platform.Collections.Stacks;
5  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
6  using Platform.Data.Doublets.Sequences.Frequencies.Counters;
7  using Platform.Data.Doublets.Sequences.Converters;
8  using Platform.Data.Doublets.Sequences.CriteriaMatchers;
9  using Platform.Data.Doublets.Sequences.Walkers;
10 using Platform.Data.Doublets.Sequences.Indexes;
11
12 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
13
14 namespace Platform.Data.Doublets.Sequences
15 {
16     public class SequencesOptions<TLink> // TODO: To use type parameter <TLink> the
17     ↪ ILinks<TLink> must contain GetConstants function.
18     {
19         private static readonly EqualityComparer<TLink> _equalityComparer =
20         ↪ EqualityComparer<TLink>.Default;
21
22         public TLink SequenceMarkerLink { get; set; }
23         public bool UseCascadeUpdate { get; set; }
24         public bool UseCascadeDelete { get; set; }
25         public bool UseIndex { get; set; } // TODO: Update Index on sequence update/delete.
26         public bool UseSequenceMarker { get; set; }
27         public bool UseCompression { get; set; }
28         public bool UseGarbageCollection { get; set; }
29         public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting { get; set; }
30         public bool EnforceSingleSequenceVersionOnWriteBasedOnNew { get; set; }
31
32         public MarkedSequenceCriterionMatcher<TLink> MarkedSequenceMatcher { get; set; }
33         public IConverter<IList<TLink>, TLink> LinksToSequenceConverter { get; set; }
34         public ISequenceIndex<TLink> Index { get; set; }
35         public ISequenceWalker<TLink> Walker { get; set; }
36         public bool ReadFullSequence { get; set; }
37
38         // TODO: Реализовать компактификацию при чтении
39         //public bool EnforceSingleSequenceVersionOnRead { get; set; }
40         //public bool UseRequestMarker { get; set; }
41         //public bool StoreRequestResults { get; set; }
42
43         public void InitOptions(ISynchronizedLinks<TLink> links)
44         {
45             if (UseSequenceMarker)
46             {
47                 if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
48                 {
49                     SequenceMarkerLink = links.CreatePoint();
50                 }
51                 else
52                 {
53                     if (!links.Exists(SequenceMarkerLink))
54                     {
55                         var link = links.CreatePoint();
56                         if (!_equalityComparer.Equals(link, SequenceMarkerLink))
57                         {
58                             throw new InvalidOperationException("Cannot recreate sequence marker
59                                 ↪ link.");
60                         }
61                     }
62                 }
63             }
64             if (MarkedSequenceMatcher == null)
65             {
66                 MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
67                     ↪ SequenceMarkerLink);
68             }
69         }
70     }
71 }

```



```

63     }
64 }
65 var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
66 if (UseCompression)
67 {
68     if (LinksToSequenceConverter == null)
69     {
70         ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
71         if (UseSequenceMarker)
72         {
73             totalSequenceSymbolFrequencyCounter = new
74                 ↪ TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
75                 ↪ MarkedSequenceMatcher);
76         }
77         else
78         {
79             totalSequenceSymbolFrequencyCounter = new
80                 ↪ TotalSequenceSymbolFrequencyCounter<TLink>(links);
81         }
82         var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
83             ↪ totalSequenceSymbolFrequencyCounter);
84         var compressingConverter = new CompressingConverter<TLink>(links,
85             ↪ balancedVariantConverter, doubletFrequenciesCache);
86         LinksToSequenceConverter = compressingConverter;
87     }
88 }
89 else
90 {
91     if (LinksToSequenceConverter == null)
92     {
93         LinksToSequenceConverter = balancedVariantConverter;
94     }
95 }
96 if (UseIndex && Index == null)
97 {
98     Index = new SequenceIndex<TLink>(links);
99 }
100 if (Walker == null)
101 {
102     Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
103 }
104 }
105 }
106
107 public void ValidateOptions()
108 {
109     if (UseGarbageCollection && !UseSequenceMarker)
110     {
111         throw new NotSupportedException("To use garbage collection UseSequenceMarker
112             ↪ option must be on.");
113     }
114 }
115 }
116 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences
7  {
8      public class SetFiller<TElement, TReturnConstant>
9      {
10         protected readonly ISet<TElement> _set;
11         protected readonly TReturnConstant _returnConstant;
12
13         public SetFiller(ISet<TElement> set, TReturnConstant returnConstant)
14         {
15             _set = set;
16             _returnConstant = returnConstant;
17         }
18
19         public SetFiller(ISet<TElement> set) : this(set, default) { }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public void Add(TElement element) => _set.Add(element);
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

25     public bool AddAndReturnTrue(TElement element)
26     {
27         _set.Add(element);
28         return true;
29     }
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     public bool AddFirstAndReturnTrue(ICollection<TElement> collection)
33     {
34         _set.Add(collection[0]);
35         return true;
36     }
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     public TReturnConstant AddAndReturnConstant(TElement element)
40     {
41         _set.Add(element);
42         return _returnConstant;
43     }
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     public TReturnConstant AddFirstAndReturnConstant(ICollection<TElement> collection)
47     {
48         _set.Add(collection[0]);
49         return _returnConstant;
50     }
51 }
52 }

```

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

```

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

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Collections.Stacks;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Sequences.Walkers
9  {
10     public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
11     {
12         public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
13             ↪ isElement) : base(links, stack, isElement) { }
14
15         public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links, stack,
16             ↪ links.IsPartialPoint) { }
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override TLink GetNextElementAfterPop(TLink element) =>
20             ↪ Links.GetSource(element);
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override TLink GetNextElementAfterPush(TLink element) =>
24             ↪ Links.GetTarget(element);
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override IEnumerable<TLink> WalkContents(TLink element)
28         {
29             var parts = Links.GetLink(element);
30             var start = Links.Constants.IndexPart + 1;
31             for (var i = parts.Count - 1; i >= start; i--)
32             {
33                 var part = parts[i];
34                 if (IsElement(part))
35                 {
36                     yield return part;
37                 }
38             }
39         }
40     }
41 }

```

```

33     }
34 }
35 }
36 }
37 }

```

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

```

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

```

```

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

```

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

```

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

```

```

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

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Collections.Stacks;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Sequences.Walkers
9  {
10     public abstract class SequenceWalkerBase<TLink> : LinksOperatorBase<TLink>,
11         ↳ ISequenceWalker<TLink>
12     {
13         private readonly IStack<TLink> _stack;
14         private readonly Func<TLink, bool> _isElement;
15
16         protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
17             ↳ isElement) : base(links)
18         {
19             _stack = stack;
20             _isElement = isElement;
21         }
22
23         protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : this(links,
24             ↳ stack, links.IsPartialPoint)
25         {
26         }
27
28         public IEnumerable<TLink> Walk(TLink sequence)
29         {
30             _stack.Clear();
31             var element = sequence;
32             if (IsElement(element))
33             {
34                 yield return element;
35             }
36             else
37             {
38                 while (true)
39                 {
40                     if (IsElement(element))
41                     {
42                         if (_stack.IsEmpty)
43                         {
44                             break;
45                         }
46                         element = _stack.Pop();
47                         foreach (var output in WalkContents(element))
48                         {
49                             yield return output;
50                         }
51                         element = GetNextElementAfterPop(element);
52                     }
53                     else
54                     {
55                         _stack.Push(element);
56                         element = GetNextElementAfterPush(element);
57                     }
58                 }
59             }
60         }
61     }
62 }

```

[MethodImpl(MethodImplOptions.AggressiveInlining)]

```

60     protected virtual bool IsElement(TLink elementLink) => _isElement(elementLink);
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     protected abstract TLink GetNextElementAfterPop(TLink element);
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     protected abstract TLink GetNextElementAfterPush(TLink element);
67
68     [MethodImpl(MethodImplOptions.AggressiveInlining)]
69     protected abstract IEnumerable<TLink> WalkContents(TLink element);
70 }
71 }

```

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

```

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

```

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

```

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

```

./Platform.Data.Doublets/SynchronizedLinks.cs

```
1 using System;
2 using System.Collections.Generic;
3 using Platform.Data.Doublets;
4 using Platform.Threading.Synchronization;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets
9 {
10     /// <remarks>
11     /// TODO: Autogeneration of synchronized wrapper (decorator).
12     /// TODO: Try to unfold code of each method using IL generation for performance improvements.
13     /// TODO: Or even to unfold multiple layers of implementations.
14     /// </remarks>
15     public class SynchronizedLinks<TLinkAddress> : ISynchronizedLinks<TLinkAddress>
16     {
17         public LinksConstants<TLinkAddress> Constants { get; }
18         public ISynchronization SyncRoot { get; }
19         public ILinks<TLinkAddress> Sync { get; }
20         public ILinks<TLinkAddress> Unsync { get; }
21
22         public SynchronizedLinks(ILinks<TLinkAddress> links) : this(new
            ↳ ReaderWriterLockSynchronization(), links) { }
23
24         public SynchronizedLinks(ISynchronization synchronization, ILinks<TLinkAddress> links)
25         {
26             SyncRoot = synchronization;
27             Sync = this;
28             Unsync = links;
29             Constants = links.Constants;
30         }
31
32         public TLinkAddress Count(IList<TLinkAddress> restriction) =>
            ↳ SyncRoot.ExecuteReadOperation(restriction, Unsync.Count);
33         public TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler,
            ↳ IList<TLinkAddress> restrictions) => SyncRoot.ExecuteReadOperation(handler,
            ↳ restrictions, (handler1, restrictions1) => Unsync.Each(handler1, restrictions1));
34         public TLinkAddress Create(IList<TLinkAddress> restrictions) =>
            ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Create);
35         public TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress>
            ↳ substitution) => SyncRoot.ExecuteWriteOperation(restrictions, substitution,
            ↳ Unsync.Update);
36         public void Delete(IList<TLinkAddress> restrictions) =>
            ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Delete);
37
38         //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
39         //↳ IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
40         //{
41         //    if (restriction != null && substitution != null &&
42         //        ↳ !substitution.EqualTo(restriction))
43         //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
44         //        ↳ substitution, substitutedHandler, Unsync.Trigger);
45         //}
46     }
```

./Platform.Data.Doublets/UInt64Link.cs

```
1 using System;
2 using System.Collections;
3 using System.Collections.Generic;
4 using Platform.Exceptions;
5 using Platform.Ranges;
6 using Platform.Singletons;
7 using Platform.Collections.Lists;
8
9 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets
12 {
13     /// <summary>
14     /// Структура описывающая уникальную связь.
15     /// </summary>
16     public struct UInt64Link : IEquatable<UInt64Link>, IReadOnlyList<ulong>, IList<ulong>
17     {
```

```

18 private static readonly LinksConstants<ulong> _constants =
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 public static implicit operator UInt64Link(ulong[] linkArray) => new
87     ↳ UInt64Link(linkArray);
88
89 public override string ToString() => Index == _constants.Null ? ToString(Source, Target)
90     ↳ : ToString(Index, Source, Target);
91
92 #region IList
93
94 public ulong this[int index]
95 {
96     get
97     {

```



```

86         Ensure.OnDebug.ArgumentInRange(index, new Range<int>(0, Length - 1),
87             ↳ nameof(index));
88         if (index == _constants.IndexPart)
89         {
90             return Index;
91         }
92         if (index == _constants.SourcePart)
93         {
94             return Source;
95         }
96         if (index == _constants.TargetPart)
97         {
98             return Target;
99         }
100         throw new NotSupportedException(); // Impossible path due to
101         ↳ Ensure.ArgumentInRange
102     }
103     set => throw new NotSupportedException();
104 }
105
106 public int Count => Length;
107
108 public bool IsReadOnly => true;
109
110 IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
111
112 public IEnumerator<ulong> GetEnumerator()
113 {
114     yield return Index;
115     yield return Source;
116     yield return Target;
117 }
118
119 public void Add(ulong item) => throw new NotSupportedException();
120
121 public void Clear() => throw new NotSupportedException();
122
123 public bool Contains(ulong item) => IndexOf(item) >= 0;
124
125 public void CopyTo(ulong[] array, int arrayIndex)
126 {
127     Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
128     Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
129         ↳ nameof(arrayIndex));
130     if (arrayIndex + Length > array.Length)
131     {
132         throw new ArgumentException();
133     }
134     array[arrayIndex++] = Index;
135     array[arrayIndex++] = Source;
136     array[arrayIndex] = Target;
137 }
138
139 public bool Remove(ulong item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
140
141 public int IndexOf(ulong item)
142 {
143     if (Index == item)
144     {
145         return _constants.IndexPart;
146     }
147     if (Source == item)
148     {
149         return _constants.SourcePart;
150     }
151     if (Target == item)
152     {
153         return _constants.TargetPart;
154     }
155     return -1;
156 }
157
158 public void Insert(int index, ulong item) => throw new NotSupportedException();
159
160 public void RemoveAt(int index) => throw new NotSupportedException();
161 #endregion
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         public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
51         {
52             if (sequence == null)
53             {
54                 return false;
55             }
56             var constants = links.Constants;
57             for (var i = 0; i < sequence.Length; i++)
58             {
59                 if (sequence[i] == constants.Any)
60                 {
61                     return true;
62                 }
63             }
64         }
65     }
66 }
```

```

62     return false;
63 }
64
65 public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
↳ Func<UInt64Link, bool> isElement, bool renderIndex = false, bool renderDebug = false)
66 {
67     var sb = new StringBuilder();
68     var visited = new HashSet<ulong>();
69     links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
↳ innerSb.Append(link.Index), renderIndex, renderDebug);
70     return sb.ToString();
71 }
72
73 public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
↳ Func<UInt64Link, bool> isElement, Action<StringBuilder, UInt64Link> appendElement,
↳ bool renderIndex = false, bool renderDebug = false)
74 {
75     var sb = new StringBuilder();
76     var visited = new HashSet<ulong>();
77     links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
↳ renderDebug);
78     return sb.ToString();
79 }
80
81 public static void AppendStructure(this ILinks<ulong> links, StringBuilder sb,
↳ HashSet<ulong> visited, ulong linkIndex, Func<UInt64Link, bool> isElement,
↳ Action<StringBuilder, UInt64Link> appendElement, bool renderIndex = false, bool
↳ renderDebug = false)
82 {
83     if (sb == null)
84     {
85         throw new ArgumentNullException(nameof(sb));
86     }
87     if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
↳ Constants.Itself)
88     {
89         return;
90     }
91     if (links.Exists(linkIndex))
92     {
93         if (visited.Add(linkIndex))
94         {
95             sb.Append('(');
96             var link = new UInt64Link(links.GetLink(linkIndex));
97             if (renderIndex)
98             {
99                 sb.Append(link.Index);
100                 sb.Append(':');
101             }
102             if (link.Source == link.Index)
103             {
104                 sb.Append(link.Index);
105             }
106             else
107             {
108                 var source = new UInt64Link(links.GetLink(link.Source));
109                 if (isElement(source))
110                 {
111                     appendElement(sb, source);
112                 }
113                 else
114                 {
115                     links.AppendStructure(sb, visited, source.Index, isElement,
↳ appendElement, renderIndex);
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         sb.Append(' ');
136     }
137     else
138     {
139         if (renderDebug)
140         {
141             sb.Append('*');
142         }
143         sb.Append(linkIndex);
144     }
145 }
146 else
147 {
148     if (renderDebug)
149     {
150         sb.Append('~');
151     }
152     sb.Append(linkIndex);
153 }
154 }
155 }
156 }

```

./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs

```

1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4  using System.IO;
5  using System.Runtime.CompilerServices;
6  using System.Threading;
7  using System.Threading.Tasks;
8  using Platform.Disposables;
9  using Platform.Timestamps;
10 using Platform.Unsafe;
11 using Platform.IO;
12 using Platform.Data.Doublets.Decorators;
13 using Platform.Exceptions;
14
15 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
16
17 namespace Platform.Data.Doublets
18 {
19     public class UInt64LinksTransactionsLayer : LinksDisposableDecoratorBase<ulong> //-V3073
20     {
21         /// <remarks>
22         /// Альтернативные варианты хранения трансформации (элемента транзакции):
23         ///
24         /// private enum TransitionType
25         /// {
26         ///     Creation,
27         ///     UpdateOf,
28         ///     UpdateTo,
29         ///     Deletion
30         /// }
31         ///
32         /// private struct Transition
33         /// {
34         ///     public ulong TransactionId;
35         ///     public UniqueTimestamp Timestamp;
36         ///     public TransactionItemType Type;
37         ///     public Link Source;
38         ///     public Link Linker;
39         ///     public Link Target;
40         /// }
41         ///
42         /// Или
43         ///
44         /// public struct TransitionHeader
45         /// {
46         ///     public ulong TransactionIdCombined;
47         ///     public ulong TimestampCombined;
48         ///
49         ///     public ulong TransactionId

```

```

50     /// {
51     ///     get
52     ///     {
53     ///         return (ulong) mask & TransactionIdCombined;
54     ///     }
55     /// }
56     ///
57     public UniqueTimestamp Timestamp
58     {
59     ///     get
60     ///     {
61     ///         return (UniqueTimestamp)mask & TransactionIdCombined;
62     ///     }
63     /// }
64     ///
65     public TransactionItemType Type
66     {
67     ///     get
68     ///     {
69     ///         // Использовать по одному биту из TransactionId и Timestamp,
70     ///         // для значения в 2 бита, которое представляет тип операции
71     ///         throw new NotImplementedException();
72     ///     }
73     /// }
74     /// }
75     ///
76     private struct Transition
77     {
78     ///     public TransitionHeader Header;
79     ///     public Link Source;
80     ///     public Link Linker;
81     ///     public Link Target;
82     /// }
83     ///
84     </remarks>
85     public struct Transition
86     {
87         public static readonly long Size = Structure<Transition>.Size;
88
89         public readonly ulong TransactionId;
90         public readonly UInt64Link Before;
91         public readonly UInt64Link After;
92         public readonly Timestamp Timestamp;
93
94         public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
95             ↪ transactionId, UInt64Link before, UInt64Link after)
96         {
97             TransactionId = transactionId;
98             Before = before;
99             After = after;
100             Timestamp = uniqueTimestampFactory.Create();
101         }
102
103         public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
104             ↪ transactionId, UInt64Link before)
105             : this(uniqueTimestampFactory, transactionId, before, default)
106         {
107         }
108
109         public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId)
110             : this(uniqueTimestampFactory, transactionId, default, default)
111         {
112         }
113
114         public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
115             ↪ {After}";
116     }
117
118     /// <remarks>
119     /// Другие варианты реализации транзакций (атомарности):
120     /// 1. Разделение хранения значения связи ((Source Target) или (Source Linker
121     ///     ↪ Target)) и индексов.
122     /// 2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
123     ///     ↪ потребуется решить вопрос
124     ///     со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
125     ///     ↪ пересечениями идентификаторов.
126     ///
127     /// Где хранить промежуточный список транзакций?

```

```

122 ///
123 /// В оперативной памяти:
124 /// Минусы:
125 /// 1. Может усложнить систему, если она будет функционировать самостоятельно,
126 /// так как нужно отдельно выделять память под список трансформаций.
127 /// 2. Выделенной оперативной памяти может не хватить, в том случае,
128 /// если транзакция использует слишком много трансформаций.
129 /// -> Можно использовать жёсткий диск для слишком длинных транзакций.
130 /// -> Максимальный размер списка трансформаций можно ограничить / задать
    ↪ константой.
131 /// 3. При подтверждении транзакции (Commit) все трансформации записываются разом
    ↪ создавая задержку.
132 ///
133 /// На жёстком диске:
134 /// Минусы:
135 /// 1. Длительный отклик, на запись каждой трансформации.
136 /// 2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
137 /// -> Это может решаться упаковкой/исключением дублирующих операций.
138 /// -> Также это может решаться тем, что короткие транзакции вообще
139 /// не будут записываться в случае отката.
140 /// 3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
    ↪ операции (трансформации)
    ↪ будут записаны в лог.
141 ///
142 ///
143 </remarks>
144 public class Transaction : DisposableBase
145 {
146     private readonly Queue<Transition> _transitions;
147     private readonly UInt64LinksTransactionsLayer _layer;
148     public bool IsCommitted { get; private set; }
149     public bool IsReverted { get; private set; }
150
151     public Transaction(UInt64LinksTransactionsLayer layer)
152     {
153         _layer = layer;
154         if (_layer._currentTransactionId != 0)
155         {
156             throw new NotSupportedException("Nested transactions not supported.");
157         }
158         IsCommitted = false;
159         IsReverted = false;
160         _transitions = new Queue<Transition>();
161         SetCurrentTransaction(layer, this);
162     }
163
164     public void Commit()
165     {
166         EnsureTransactionAllowsWriteOperations(this);
167         while (_transitions.Count > 0)
168         {
169             var transition = _transitions.Dequeue();
170             _layer._transitions.Enqueue(transition);
171         }
172         _layer._lastCommittedTransactionId = _layer._currentTransactionId;
173         IsCommitted = true;
174     }
175
176     private void Revert()
177     {
178         EnsureTransactionAllowsWriteOperations(this);
179         var transitionsToRevert = new Transition[_transitions.Count];
180         _transitions.CopyTo(transitionsToRevert, 0);
181         for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
182         {
183             _layer.RevertTransition(transitionsToRevert[i]);
184         }
185         IsReverted = true;
186     }
187
188     public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
    ↪ Transaction transaction)
189     {
190         layer._currentTransactionId = layer._lastCommittedTransactionId + 1;
191         layer._currentTransactionTransitions = transaction._transitions;
192         layer._currentTransaction = transaction;
193     }
194
195     public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
196     {

```

```

197         if (transaction.IsReverted)
198         {
199             throw new InvalidOperationException("Transation is reverted.");
200         }
201         if (transaction.IsCommitted)
202         {
203             throw new InvalidOperationException("Transation is committed.");
204         }
205     }
206
207     protected override void Dispose(bool manual, bool wasDisposed)
208     {
209         if (!wasDisposed && _layer != null && !_layer.IsDisposed)
210         {
211             if (!IsCommitted && !IsReverted)
212             {
213                 Revert();
214             }
215             _layer.ResetCurrentTransation();
216         }
217     }
218 }
219
220 public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
221
222 private readonly string _logAddress;
223 private readonly FileStream _log;
224 private readonly Queue<Transition> _transitions;
225 private readonly UniqueTimestampFactory _uniqueTimestampFactory;
226 private Task _transitionsPusher;
227 private Transition _lastCommittedTransition;
228 private ulong _currentTransactionId;
229 private Queue<Transition> _currentTransactionTransitions;
230 private Transaction _currentTransaction;
231 private ulong _lastCommittedTransactionId;
232
233 public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
234     : base(links)
235 {
236     if (string.IsNullOrEmpty(logAddress))
237     {
238         throw new ArgumentNullException(nameof(logAddress));
239     }
240     // В первой строке файла хранится последняя законченную транзакцию.
241     // При запуске это используется для проверки удачного закрытия файла лога.
242     // In the first line of the file the last committed transaction is stored.
243     // On startup, this is used to check that the log file is successfully closed.
244     var lastCommittedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
245     var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
246     if (!lastCommittedTransition.Equals(lastWrittenTransition))
247     {
248         Dispose();
249         throw new NotSupportedException("Database is damaged, autorecovery is not
250             ↳ supported yet.");
251     }
252     if (lastCommittedTransition.Equals(default(Transition)))
253     {
254         FileHelpers.WriteFirst(logAddress, lastCommittedTransition);
255     }
256     _lastCommittedTransition = lastCommittedTransition;
257     // TODO: Think about a better way to calculate or store this value
258     var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
259     _lastCommittedTransactionId = allTransitions.Max(x => x.TransactionId);
260     _uniqueTimestampFactory = new UniqueTimestampFactory();
261     _logAddress = logAddress;
262     _log = FileHelpers.Append(logAddress);
263     _transitions = new Queue<Transition>();
264     _transitionsPusher = new Task(TransitionsPusher);
265     _transitionsPusher.Start();
266 }
267
268 public IList<ulong> GetLinkValue(ulong link) => Links.GetLink(link);
269
270 public override ulong Create(IList<ulong> restrictions)
271 {
272     var createdLinkIndex = Links.Create();
273     var createdLink = new UInt64Link(Links.GetLink(createdLinkIndex));
274     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
275         ↳ default, createdLink));

```

```

274         return createdLinkIndex;
275     }
276
277     public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
278     {
279         var linkIndex = restrictions[Constants.IndexPart];
280         var beforeLink = new UInt64Link(Links.GetLink(linkIndex));
281         linkIndex = Links.Update(restrictions, substitution);
282         var afterLink = new UInt64Link(Links.GetLink(linkIndex));
283         CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
284             ↪ beforeLink, afterLink));
285         return linkIndex;
286     }
287
288     public override void Delete(IList<ulong> restrictions)
289     {
290         var link = restrictions[Constants.IndexPart];
291         var deletedLink = new UInt64Link(Links.GetLink(link));
292         Links.Delete(link);
293         CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
294             ↪ deletedLink, default));
295     }
296
297     [MethodImpl(MethodImplOptions.AggressiveInlining)]
298     private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
299         ↪ _transitions;
300
301     private void CommitTransition(Transition transition)
302     {
303         if (_currentTransaction != null)
304         {
305             Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
306         }
307         var transitions = GetCurrentTransitions();
308         transitions.Enqueue(transition);
309     }
310
311     private void RevertTransition(Transition transition)
312     {
313         if (transition.After.IsNull()) // Revert Deletion with Creation
314         {
315             Links.Create();
316         }
317         else if (transition.Before.IsNull()) // Revert Creation with Deletion
318         {
319             Links.Delete(transition.After.Index);
320         }
321         else // Revert Update
322         {
323             Links.Update(new[] { transition.After.Index, transition.Before.Source,
324                 ↪ transition.Before.Target });
325         }
326     }
327
328     private void ResetCurrentTransation()
329     {
330         _currentTransactionId = 0;
331         _currentTransactionTransitions = null;
332         _currentTransaction = null;
333     }
334
335     private void PushTransitions()
336     {
337         if (_log == null || _transitions == null)
338         {
339             return;
340         }
341         for (var i = 0; i < _transitions.Count; i++)
342         {
343             var transition = _transitions.Dequeue();
344
345             _log.Write(transition);
346             _lastCommittedTransition = transition;
347         }
348     }
349
350     private void TransitionsPusher()
351     {
352         while (!IsDisposed && _transitionsPusher != null)

```



```

349         {
350             Thread.Sleep(DefaultPushDelay);
351             PushTransitions();
352         }
353     }
354
355     public Transaction BeginTransaction() => new Transaction(this);
356
357     private void DisposeTransitions()
358     {
359         try
360         {
361             var pusher = _transitionsPusher;
362             if (pusher != null)
363             {
364                 _transitionsPusher = null;
365                 pusher.Wait();
366             }
367             if (_transitions != null)
368             {
369                 PushTransitions();
370             }
371             _log.DisposeIfPossible();
372             FileHelpers.WriteFirst(_logAddress, _lastCommittedTransition);
373         }
374         catch (Exception ex)
375         {
376             ex.Ignore();
377         }
378     }
379
380     #region DisposalBase
381
382     protected override void Dispose(bool manual, bool wasDisposed)
383     {
384         if (!wasDisposed)
385         {
386             DisposeTransitions();
387         }
388         base.Dispose(manual, wasDisposed);
389     }
390
391     #endregion
392 }
393

```

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

```

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

```

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

```

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

```

```

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

```

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

```

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

```

```

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

```

```

118     }
119
120     public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
121     {
122         var result = new List<ulong[]>();
123         var offset = 0;
124         while (offset < sequence.Length)
125         {
126             var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
127             var relativeLength = 1;
128             var absoluteLength = offset + relativeLength;
129             while (absoluteLength < sequence.Length &&
130                 currentCategory ==
131                 CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
132             {
133                 relativeLength++;
134                 absoluteLength++;
135             }
136             // char array to ulong array
137             var innerSequence = new ulong[relativeLength];
138             var maxLength = offset + relativeLength;
139             for (var i = offset; i < maxLength; i++)
140             {
141                 innerSequence[i - offset] = FromCharToLink(sequence[i]);
142             }
143             result.Add(innerSequence);
144             offset += relativeLength;
145         }
146         return result;
147     }
148
149     public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
150     {
151         var result = new List<ulong[]>();
152         var offset = 0;
153         while (offset < array.Length)
154         {
155             var relativeLength = 1;
156             if (array[offset] <= LastCharLink)
157             {
158                 var currentCategory =
159                 CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
160                 var absoluteLength = offset + relativeLength;
161                 while (absoluteLength < array.Length &&
162                     array[absoluteLength] <= LastCharLink &&
163                     currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(
164                     CharUnicodeInfo.GetUnicodeCategory(array[absoluteLength])))
165                 {
166                     relativeLength++;
167                     absoluteLength++;
168                 }
169             }
170             else
171             {
172                 var absoluteLength = offset + relativeLength;
173                 while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
174                 {
175                     relativeLength++;
176                     absoluteLength++;
177                 }
178             }
179             // copy array
180             var innerSequence = new ulong[relativeLength];
181             var maxLength = offset + relativeLength;
182             for (var i = offset; i < maxLength; i++)
183             {
184                 innerSequence[i - offset] = array[i];
185             }
186             result.Add(innerSequence);
187             offset += relativeLength;
188         }
189         return result;
190     }
191 }
192 }

```

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

```

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

```

```

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

```

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

```

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

```

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

```

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

```

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

```
1 using System;
2 using Platform.Interfaces;
3 using Platform.Numbers;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Unicode
8 {
9     public class UnicodeSymbolToCharConverter<TLink> : LinksOperatorBase<TLink>,
10     ↪ IConverter<TLink, char>
11     {
12         private readonly IConverter<TLink> _numberToAddressConverter;
13         private readonly ICriterionMatcher<TLink> _unicodeSymbolCriterionMatcher;
14
15         public UnicodeSymbolToCharConverter(ILinks<TLink> links, IConverter<TLink>
16         ↪ numberToAddressConverter, ICriterionMatcher<TLink> unicodeSymbolCriterionMatcher) :
17         ↪ base(links)
18         {
19             _numberToAddressConverter = numberToAddressConverter;
20             _unicodeSymbolCriterionMatcher = unicodeSymbolCriterionMatcher;
21
22             public char Convert(TLink source)
23             {
24                 if (!_unicodeSymbolCriterionMatcher.IsMatched(source))
25                 {
26                     throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
27                     ↪ not a unicode symbol.");
28                 }
29                 return (char)(ushort)(Integer<TLink>)_numberToAddressConverter.Convert(Links.GetSour
30                 ↪ ce(source));
31             }
32         }
33     }
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 GreaterOrEqualPerfomanceTest()
19         {
20             const int N = 1000000;
21
22             ulong x = 10;
23             ulong y = 500;
24
25             bool result = false;
26
27             var ts1 = Performance.Measure(() =>
28             {
29                 for (int i = 0; i < N; i++)
30                 {
31                     result = Compare(x, y) >= 0;
32                 }
33             });
34
35             var comparer1 = Comparer<ulong>.Default;
36
37             var ts2 = Performance.Measure(() =>
38             {
39                 for (int i = 0; i < N; i++)
40                 {
41                     result = comparer1.Compare(x, y) >= 0;
42                 }
43             });
44         }
45     }
46 }
```

```

44
45     Func<ulong, ulong, int> compareReference = comparer1.Compare;
46
47     var ts3 = Performance.Measure(() =>
48     {
49         for (int i = 0; i < N; i++)
50         {
51             result = compareReference(x, y) >= 0;
52         }
53     });
54
55     var comparer2 = new UInt64Comparer();
56
57     var ts4 = Performance.Measure(() =>
58     {
59         for (int i = 0; i < N; i++)
60         {
61             result = comparer2.Compare(x, y) >= 0;
62         }
63     });
64
65     Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {result}");
66 }
67 }
68 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using Xunit;
4  using Platform.Diagnostics;
5
6  namespace Platform.Data.Doublets.Tests
7  {
8      public static class EqualityTests
9      {
10         protected class UInt64EqualityComparer : IEqualityComparer<ulong>
11         {
12             public bool Equals(ulong x, ulong y) => x == y;
13
14             public int GetHashCode(ulong obj) => obj.GetHashCode();
15         }
16
17         private static bool Equals1<T>(T x, T y) => Equals(x, y);
18
19         private static bool Equals2<T>(T x, T y) => x.Equals(y);
20
21         private static bool Equals3(ulong x, ulong y) => x == y;
22
23         [Fact]
24         public static void EqualsPerfomanceTest()
25         {
26             const int N = 1000000;
27
28             ulong x = 10;
29             ulong y = 500;
30
31             bool result = false;
32
33             var ts1 = Performance.Measure(() =>
34             {
35                 for (int i = 0; i < N; i++)
36                 {
37                     result = Equals1(x, y);
38                 }
39             });
40
41             var ts2 = Performance.Measure(() =>
42             {
43                 for (int i = 0; i < N; i++)
44                 {
45                     result = Equals2(x, y);
46                 }
47             });
48
49             var ts3 = Performance.Measure(() =>
50             {
51                 for (int i = 0; i < N; i++)
52                 {
53                     result = Equals3(x, y);

```

```

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

```

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

```

1  using System;
2  using Xunit;
3  using Platform.Reflection;
4  using Platform.Memory;
5  using Platform.Scopes;
6  using Platform.Data.Doublets.ResizableDirectMemory;
7
8  namespace Platform.Data.Doublets.Tests
9  {
10     public unsafe static class GenericLinksTests
11     {
12         [Fact]
13         public static void CRUDTest()
14         {
15             Using<byte>(links => links.TestCRUDOperations());
16             Using<ushort>(links => links.TestCRUDOperations());
17             Using<uint>(links => links.TestCRUDOperations());
18             Using<ulong>(links => links.TestCRUDOperations());
19         }
20
21         [Fact]
22         public static void RawNumbersCRUDTest()
23         {
24             Using<byte>(links => links.TestRawNumbersCRUDOperations());
25             Using<ushort>(links => links.TestRawNumbersCRUDOperations());
26             Using<uint>(links => links.TestRawNumbersCRUDOperations());

```



```

27     Using<ulong>(links => links.TestRawNumbersCRUDOperations());
28 }
29
30 [Fact]
31 public static void MultipleRandomCreationsAndDeletionsTest()
32 {
33     //if (!RuntimeInformation.IsOSPlatform(OSPlatform.Linux))
34     //{
35         //    Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution(
36             ↪ ).TestMultipleRandomCreationsAndDeletions(16)); // Cannot use more because
37             ↪ current implementation of tree cuts out 5 bits from the address space.
38         //    Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolutio
39             ↪ n().TestMultipleRandomCreationsAndDeletions(100));
40         //    Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution(
41             ↪ ).TestMultipleRandomCreationsAndDeletions(100));
42         //}
43     Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
44         ↪ tMultipleRandomCreationsAndDeletions(100));
45 }
46
47 private static void Using<TLink>(Action<ILinks<TLink>> action)
48 {
49     using (var scope = new Scope<Types<HeapResizableDirectMemory,
50         ↪ ResizableDirectMemoryLinks<TLink>>>())
51     {
52         action(scope.Use<ILinks<TLink>>());
53     }
54 }
55 }

```

./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);

```

```

44     var linkToItsFrequencyNumberConverter = new
        ↳ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
        ↳ unaryNumberToAddressConverter);
45     var sequenceToItsLocalElementLevelsConverter = new
        ↳ SequenceToItsLocalElementLevelsConverter<ulong>(links,
        ↳ linkToItsFrequencyNumberConverter);
46     var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
        ↳ sequenceToItsLocalElementLevelsConverter);
47
48     var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
        ↳ Walker = new LeveledSequenceWalker<ulong>(links) });
49
50     ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
        ↳ index, optimalVariantConverter);
51 }
52 }
53
54 [Fact]
55 public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
56 {
57     using (var scope = new TempLinksTestScope(useSequences: false))
58     {
59         var links = scope.Links;
60
61         links.UseUnicode();
62
63         var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);
64
65         var linksToFrequencies = new Dictionary<ulong, ulong>();
66
67         var totalSequenceSymbolFrequencyCounter = new
        ↳ TotalSequenceSymbolFrequencyCounter<ulong>(links);
68
69         var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
        ↳ totalSequenceSymbolFrequencyCounter);
70
71         var index = new
        ↳ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
72         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache);
73
74         var sequenceToItsLocalElementLevelsConverter = new
        ↳ SequenceToItsLocalElementLevelsConverter<ulong>(links,
        ↳ linkToItsFrequencyNumberConverter);
75         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
        ↳ sequenceToItsLocalElementLevelsConverter);
76
77         var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
        ↳ Walker = new LeveledSequenceWalker<ulong>(links) });
78
79         ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
        ↳ index, optimalVariantConverter);
80     }
81 }
82
83 private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
    ↳ SequenceToItsLocalElementLevelsConverter<ulong>
    ↳ sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
    ↳ OptimalVariantConverter<ulong> optimalVariantConverter)
84 {
85     index.Add(sequence);
86
87     var optimalVariant = optimalVariantConverter.Convert(sequence);
88
89     var readSequence1 = sequences.ToList(optimalVariant);
90
91     Assert.True(sequence.SequenceEqual(readSequence1));
92 }
93 }
94 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Diagnostics;
4 using System.Linq;
5 using Xunit;
6 using Platform.Data.Sequences;
7 using Platform.Data.Doublets.Sequences.Converters;

```

```

8 using Platform.Data.Doublets.Sequences.Walkers;
9 using Platform.Data.Doublets.Sequences;
10
11 namespace Platform.Data.Doublets.Tests
12 {
13     public static class ReadSequenceTests
14     {
15         [Fact]
16         public static void ReadSequenceTest()
17         {
18             const long sequenceLength = 2000;
19
20             using (var scope = new TempLinksTestScope(useSequences: false))
21             {
22                 var links = scope.Links;
23                 var sequences = new Sequences.Sequences(links, new SequencesOptions

```

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

```

1 using System.IO;
2 using Xunit;
3 using Platform.Singletons;
4 using Platform.Memory;
5 using Platform.Data.Doublets.ResizableDirectMemory;
6
7 namespace Platform.Data.Doublets.Tests
8 {
9     public static class ResizableDirectMemoryLinksTests
10     {
11         private static readonly LinksConstants

```

```

20     }
21     File.Delete(tempFilename);
22 }
23
24 [Fact]
25 public static void BasicHeapMemoryTest()
26 {
27     using (var memory = new
28         ↪ HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
29     using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
30         ↪ UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
31     {
32         memoryAdapter.TestBasicMemoryOperations();
33     }
34
35 private static void TestBasicMemoryOperations(this ILinks<ulong> memoryAdapter)
36 {
37     var link = memoryAdapter.Create();
38     memoryAdapter.Delete(link);
39 }
40
41 [Fact]
42 public static void NonexistentReferencesHeapMemoryTest()
43 {
44     using (var memory = new
45         ↪ HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
46     using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
47         ↪ UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
48     {
49         memoryAdapter.TestNonexistentReferences();
50     }
51
52 private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
53 {
54     var link = memoryAdapter.Create();
55     memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
56     var resultLink = _constants.Null;
57     memoryAdapter.Each(foundLink =>
58     {
59         resultLink = foundLink[_constants.IndexPart];
60         return _constants.Break;
61     }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
62     Assert.True(resultLink == link);
63     Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
64     memoryAdapter.Delete(link);
65 }
66 }
67 }

```

./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))
35             {
36                 var links = scope.Links;
37                 var sequences = scope.Sequences;
38
39                 var sequence = new ulong[sequenceLength];
40                 for (var i = 0; i < sequenceLength; i++)
41                 {
42                     sequence[i] = links.Create();
43                 }
44
45                 var sw1 = Stopwatch.StartNew();
46                 var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
47
48                 var sw2 = Stopwatch.StartNew();
49                 var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();

```

```

49         Assert.True(results1.Count > results2.Length);
50         Assert.True(sw1.Elapsed > sw2.Elapsed);
51
52         for (var i = 0; i < sequenceLength; i++)
53         {
54             links.Delete(sequence[i]);
55         }
56
57         Assert.True(links.Count() == 0);
58     }
59 }
60
61 // [Fact]
62 // public void CUDTest()
63 // {
64 //     var tempFilename = Path.GetTempFileName();
65 //
66 //     const long sequenceLength = 8;
67 //
68 //     const ulong itself = LinksConstants.Itself;
69 //
70 //     using (var memoryAdapter = new ResizableDirectMemoryLinks(tempFilename,
71 //         ↪ DefaultLinksSizeStep))
72 //     using (var links = new Links(memoryAdapter))
73 //     {
74 //         var sequence = new ulong[sequenceLength];
75 //         for (var i = 0; i < sequenceLength; i++)
76 //             sequence[i] = links.Create(itself, itself);
77 //
78 //         SequencesOptions o = new SequencesOptions();
79 //
80 //         TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
81 //             o.
82 //
83 //         var sequences = new Sequences(links);
84 //
85 //         var sw1 = Stopwatch.StartNew();
86 //         var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
87 //
88 //         var sw2 = Stopwatch.StartNew();
89 //         var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
90 //
91 //         Assert.True(results1.Count > results2.Length);
92 //         Assert.True(sw1.Elapsed > sw2.Elapsed);
93 //
94 //         for (var i = 0; i < sequenceLength; i++)
95 //             links.Delete(sequence[i]);
96 //     }
97 //
98 //     File.Delete(tempFilename);
99 // }
100
101 [Fact]
102 public static void AllVariantsSearchTest()
103 {
104     const long sequenceLength = 8;
105
106     using (var scope = new TempLinksTestScope(useSequences: true))
107     {
108         var links = scope.Links;
109         var sequences = scope.Sequences;
110
111         var sequence = new ulong[sequenceLength];
112         for (var i = 0; i < sequenceLength; i++)
113         {
114             sequence[i] = links.Create();
115         }
116
117         var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();
118
119         // for (int i = 0; i < createResults.Length; i++)
120         //     sequences.Create(createResults[i]);
121
122         var sw0 = Stopwatch.StartNew();
123         var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
124
125         var sw1 = Stopwatch.StartNew();
126         var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
127

```

```

128     var sw2 = Stopwatch.StartNew();
129     var searchResults2 = sequences.Each1(sequence); sw2.Stop();
130
131     var sw3 = Stopwatch.StartNew();
132     var searchResults3 = sequences.Each(sequence.ConvertToRestrictionsValues());
133     ↪ sw3.Stop();
134
135     var intersection0 = createResults.Intersect(searchResults0).ToList();
136     Assert.True(intersection0.Count == searchResults0.Count);
137     Assert.True(intersection0.Count == createResults.Length);
138
139     var intersection1 = createResults.Intersect(searchResults1).ToList();
140     Assert.True(intersection1.Count == searchResults1.Count);
141     Assert.True(intersection1.Count == createResults.Length);
142
143     var intersection2 = createResults.Intersect(searchResults2).ToList();
144     Assert.True(intersection2.Count == searchResults2.Count);
145     Assert.True(intersection2.Count == createResults.Length);
146
147     var intersection3 = createResults.Intersect(searchResults3).ToList();
148     Assert.True(intersection3.Count == searchResults3.Count);
149     Assert.True(intersection3.Count == createResults.Length);
150
151     for (var i = 0; i < sequenceLength; i++)
152     {
153         links.Delete(sequence[i]);
154     }
155 }
156
157 [Fact]
158 public static void BalancedVariantSearchTest()
159 {
160     const long sequenceLength = 200;
161
162     using (var scope = new TempLinksTestScope(useSequences: true))
163     {
164         var links = scope.Links;
165         var sequences = scope.Sequences;
166
167         var sequence = new ulong[sequenceLength];
168         for (var i = 0; i < sequenceLength; i++)
169         {
170             sequence[i] = links.Create();
171         }
172
173         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
174
175         var sw1 = Stopwatch.StartNew();
176         var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
177
178         var sw2 = Stopwatch.StartNew();
179         var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();
180
181         var sw3 = Stopwatch.StartNew();
182         var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();
183
184         // На количестве в 200 элементов это будет занимать вечность
185         //var sw4 = Stopwatch.StartNew();
186         //var searchResults4 = sequences.Each(sequence); sw4.Stop();
187
188         Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);
189
190         Assert.True(searchResults3.Count == 1 && balancedVariant ==
191             ↪ searchResults3.First());
192
193         //Assert.True(sw1.Elapsed < sw2.Elapsed);
194
195         for (var i = 0; i < sequenceLength; i++)
196         {
197             links.Delete(sequence[i]);
198         }
199     }
200
201 [Fact]
202 public static void AllPartialVariantsSearchTest()
203 {
204     const long sequenceLength = 8;

```

```

205
206 using (var scope = new TempLinksTestScope(useSequences: true))
207 {
208     var links = scope.Links;
209     var sequences = scope.Sequences;
210
211     var sequence = new ulong[sequenceLength];
212     for (var i = 0; i < sequenceLength; i++)
213     {
214         sequence[i] = links.Create();
215     }
216
217     var createResults = sequences.CreateAllVariants2(sequence);
218
219     //var createResultsStrings = createResults.Select(x => x + ": " +
220     ↪ sequences.FormatSequence(x)).ToList();
221     //Global.Trash = createResultsStrings;
222
223     var partialSequence = new ulong[sequenceLength - 2];
224
225     Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
226
227     var sw1 = Stopwatch.StartNew();
228     var searchResults1 =
229     ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
230
231     var sw2 = Stopwatch.StartNew();
232     var searchResults2 =
233     ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
234
235     //var sw3 = Stopwatch.StartNew();
236     //var searchResults3 =
237     ↪ sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
238
239     var sw4 = Stopwatch.StartNew();
240     var searchResults4 =
241     ↪ sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();
242
243     //Global.Trash = searchResults3;
244
245     //var searchResults1Strings = searchResults1.Select(x => x + ": " +
246     ↪ sequences.FormatSequence(x)).ToList();
247     //Global.Trash = searchResults1Strings;
248
249     var intersection1 = createResults.Intersect(searchResults1).ToList();
250     Assert.True(intersection1.Count == createResults.Length);
251
252     var intersection2 = createResults.Intersect(searchResults2).ToList();
253     Assert.True(intersection2.Count == createResults.Length);
254
255     var intersection4 = createResults.Intersect(searchResults4).ToList();
256     Assert.True(intersection4.Count == createResults.Length);
257
258     for (var i = 0; i < sequenceLength; i++)
259     {
260         links.Delete(sequence[i]);
261     }
262 }
263
264 [Fact]
265 public static void BalancedPartialVariantsSearchTest()
266 {
267     const long sequenceLength = 200;
268
269     using (var scope = new TempLinksTestScope(useSequences: true))
270     {
271         var links = scope.Links;
272         var sequences = scope.Sequences;
273
274         var sequence = new ulong[sequenceLength];
275         for (var i = 0; i < sequenceLength; i++)
276         {
277             sequence[i] = links.Create();
278         }
279
280         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
281
282         var balancedVariant = balancedVariantConverter.Convert(sequence);

```



```

279     var partialSequence = new ulong[sequenceLength - 2];
280
281     Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
282
283     var sw1 = Stopwatch.StartNew();
284     var searchResults1 =
285         ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
286
287     var sw2 = Stopwatch.StartNew();
288     var searchResults2 =
289         ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
290
291     Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
292
293     Assert.True(searchResults2.Count == 1 && balancedVariant ==
294         ↪ searchResults2.First());
295
296     for (var i = 0; i < sequenceLength; i++)
297     {
298         links.Delete(sequence[i]);
299     }
300 }
301
302 [Fact(Skip = "Correct implementation is pending")]
303 public static void PatternMatchTest()
304 {
305     var zeroOrMany = Sequences.Sequences.ZeroOrMany;
306
307     using (var scope = new TempLinksTestScope(useSequences: true))
308     {
309         var links = scope.Links;
310         var sequences = scope.Sequences;
311
312         var e1 = links.Create();
313         var e2 = links.Create();
314
315         var sequence = new[]
316         {
317             e1, e2, e1, e2 // mama / papa
318         };
319
320         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
321
322         var balancedVariant = balancedVariantConverter.Convert(sequence);
323
324         // 1: [1]
325         // 2: [2]
326         // 3: [1,2]
327         // 4: [1,2,1,2]
328
329         var doublet = links.GetSource(balancedVariant);
330
331         var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);
332
333         Assert.True(matchedSequences1.Count == 0);
334
335         var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);
336
337         Assert.True(matchedSequences2.Count == 0);
338
339         var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);
340
341         Assert.True(matchedSequences3.Count == 0);
342
343         var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
344
345         Assert.Contains(doublet, matchedSequences4);
346         Assert.Contains(balancedVariant, matchedSequences4);
347
348         for (var i = 0; i < sequence.Length; i++)
349         {
350             links.Delete(sequence[i]);
351         }
352     }
353 }
354
355 [Fact]
356 public static void IndexTest()
357 {

```

```

356 using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
    ↳ true }, useSequences: true))
357 {
358     var links = scope.Links;
359     var sequences = scope.Sequences;
360     var index = sequences.Options.Index;
361
362     var e1 = links.Create();
363     var e2 = links.Create();
364
365     var sequence = new[]
366     {
367         e1, e2, e1, e2 // mama / papa
368     };
369
370     Assert.False(index.MightContain(sequence));
371
372     index.Add(sequence);
373
374     Assert.True(index.MightContain(sequence));
375 }
376 }
377
378 /// <summary>Imported from https://raw.githubusercontent.com/wiki/Konard/LinksPlatform/%
    ↳ D0%9E-%D1%82%D0%BE%D0%BC%2C-%D0%BA%D0%B0%D0%BA-%D0%B2%D1%81%D1%91-%D0%BD%D0%B0%D1%87
    ↳ %D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>
379 private static readonly string _exampleText =
380     @"([english
    ↳ version] (https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))

```

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

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

Что может быть минимальным рисунком, образом, графикой? Может быть это точка? Это ли простейшая
 ↳ форма? Но есть ли у точки размер? Цвет? Масса? Координаты? Время существования?

![чёрное пространство, чёрная
 ↳ точка] (<https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png>
 ↳ "чёрное пространство, чёрная
 ↳ точка") (<https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png>)

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

![белая точка, чёрная
 ↳ точка] (<https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png> "белая
 ↳ точка, чёрная
 ↳ точка") (<https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png>)

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

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

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?

![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>)

```

402 Как ещё можно использовать грань, черту, линию? А что если она может что-то соединять, может
    ↳ тогда её нужно повернуть? Почему то, что перпендикулярно вертикальному горизонтально?
    ↳ Горизонт? Инвертирует ли это смысл? Что такое смысл? Из чего состоит смысл? Существует ли
    ↳ элементарная единица смысла?
403
404 [![белый круг, чёрная горизонтальная
    ↳ линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png "'белый
    ↳ круг, чёрная горизонтальная
    ↳ линия"")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png)
405
406 Соединять, допустим, а какой смысл в этом есть ещё? Что если помимо смысла "'соединить,
    ↳ связать"', есть ещё и смысл направления "'от начала к концу"'? От предка к потомку? От
    ↳ родителя к ребёнку? От общего к частному?
407
408 [![белая горизонтальная линия, чёрная горизонтальная
    ↳ стрелка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png
    ↳ "'белая горизонтальная линия, чёрная горизонтальная
    ↳ стрелка"")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png)
409
410 Шаг назад. Возьмём опять отделённую область, которая лишь та же замкнутая линия, что ещё она
    ↳ может представлять собой? Объект? Но в чём его суть? Разве не в том, что у него есть
    ↳ граница, разделяющая внутреннее и внешнее? Допустим связь, стрелка, линия соединяет два
    ↳ объекта, как бы это выглядело?
411
412 [![белая связь, чёрная направленная
    ↳ связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png "'белая
    ↳ связь, чёрная направленная
    ↳ связь"")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png)
413
414 Допустим у нас есть смысл "'связать"' и смысл "'направления"', много ли это нам даёт? Много ли
    ↳ вариантов интерпретации? А что если уточнить, каким именно образом выполнена связь? Что если
    ↳ можно задать ей чёткий, конкретный смысл? Что это будет? Тип? Глагол? Связка? Действие?
    ↳ Трансформация? Переход из состояния в состояние? Или всё это и есть объект, суть которого в
    ↳ его конечном состоянии, если конечно конец определён направлением?
415
416 [![белая обычная и направленная связи, чёрная типизированная
    ↳ связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png "'белая
    ↳ обычная и направленная связи, чёрная типизированная
    ↳ связь"")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png)
417
418 А что если всё это время, мы смотрели на суть как бы снаружи? Можно ли взглянуть на это изнутри?
    ↳ Что будет внутри объектов? Объекты ли это? Или это связи? Может ли эта структура описать
    ↳ сама себя? Но что тогда получится, разве это не рекурсия? Может это фрактал?
419
420 [![белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная типизированная
    ↳ связь с рекурсивной внутренней
    ↳ структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png
    ↳ "'белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная
    ↳ типизированная связь с рекурсивной внутренней структурой"")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png)
421
422 На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом
    ↳ рекурсии или фрактала?
423
424 [![белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная
    ↳ типизированная связь с двойной рекурсивной внутренней
    ↳ структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png
    ↳ "'белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная
    ↳ типизированная связь с двойной рекурсивной внутренней структурой"")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png)
425
426 Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы? Буквы?
    ↳ Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?
427
428 [![белая обычная и направленная связи со структурой из 8 цветных элементов последовательности,
    ↳ чёрная типизированная связь со структурой из 8 цветных элементов последовательности](https://
    ↳ /raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png "'белая обычная и
    ↳ направленная связи со структурой из 8 цветных элементов последовательности, чёрная
    ↳ типизированная связь со структурой из 8 цветных элементов последовательности"")] (https://raw
    ↳ .githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png)
429
430 ...
431
432 [![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animat
    ↳ ion-500.gif
    ↳ "'анимация"")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro
    ↳ -animation-500.gif)";
433
434 private static readonly string _exampleLoremIpsumText =

```

```

435         @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
         ↳ incididunt ut labore et dolore magna aliqua.
436 Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
         ↳ consequat.";
437
438 [Fact]
439 public static void CompressionTest()
440 {
441     using (var scope = new TempLinksTestScope(useSequences: true))
442     {
443         var links = scope.Links;
444         var sequences = scope.Sequences;
445
446         var e1 = links.Create();
447         var e2 = links.Create();
448
449         var sequence = new[]
450         {
451             e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
452         };
453
454         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links.Unsync);
455         var totalSequenceSymbolFrequencyCounter = new
456             ↳ TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
457         var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync,
458             ↳ totalSequenceSymbolFrequencyCounter);
459         var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
460             ↳ balancedVariantConverter, doubletFrequenciesCache);
461
462         var compressedVariant = compressingConverter.Convert(sequence);
463
464         // 1: [1]          (1->1) point
465         // 2: [2]          (2->2) point
466         // 3: [1,2]        (1->2) doublet
467         // 4: [1,2,1,2]    (3->3) doublet
468
469         Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
470         Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
471         Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
472         Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);
473
474         var source = _constants.SourcePart;
475         var target = _constants.TargetPart;
476
477         Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
478         Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
479         Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
480         Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);
481
482         // 4 - length of sequence
483         Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0)
484             ↳ == sequence[0]);
485         Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
486             ↳ == sequence[1]);
487         Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
488             ↳ == sequence[2]);
489         Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
490             ↳ == sequence[3]);
491     }
492 }
493
494 [Fact]
495 public static void CompressionEfficiencyTest()
496 {
497     var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
498         ↳ StringSplitOptions.RemoveEmptyEntries);
499     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
500     var totalCharacters = arrays.Select(x => x.Length).Sum();
501
502     using (var scope1 = new TempLinksTestScope(useSequences: true))
503     using (var scope2 = new TempLinksTestScope(useSequences: true))
504     using (var scope3 = new TempLinksTestScope(useSequences: true))
505     {
506         scope1.Links.Unsync.UseUnicode();
507         scope2.Links.Unsync.UseUnicode();
508         scope3.Links.Unsync.UseUnicode();
509
510         var balancedVariantConverter1 = new
511             ↳ BalancedVariantConverter<ulong>(scope1.Links.Unsync);

```

```

503     var totalSequenceSymbolFrequencyCounter = new
504         ↳ TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
505     var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
506         ↳ totalSequenceSymbolFrequencyCounter);
507     var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
508         ↳ balancedVariantConverter1, linkFrequenciesCache1,
509         ↳ doInitialFrequenciesIncrement: false);
510
511     //var compressor2 = scope2.Sequences;
512     var compressor3 = scope3.Sequences;
513
514     var constants = Default<LinksConstants<ulong>>.Instance;
515
516     var sequences = compressor3;
517     //var meaningRoot = links.CreatePoint();
518     //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
519     //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
520     //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
521         ↳ constants.Itself);
522
523     //var unaryNumberToAddressConverter = new
524         ↳ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
525     //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
526         ↳ unaryOne);
527     //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
528         ↳ frequencyMarker, unaryOne, unaryNumberIncrementer);
529     //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
530         ↳ frequencyPropertyMarker, frequencyMarker);
531     //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
532         ↳ frequencyPropertyOperator, frequencyIncrementer);
533     //var linkToItsFrequencyNumberConverter = new
534         ↳ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
535         ↳ unaryNumberToAddressConverter);
536
537     var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
538         ↳ totalSequenceSymbolFrequencyCounter);
539
540     var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache3);
541
542     var sequenceToItsLocalElementLevelsConverter = new
543         ↳ SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
544         ↳ linkToItsFrequencyNumberConverter);
545     var optimalVariantConverter = new
546         ↳ OptimalVariantConverter<ulong>(scope3.Links.Unsync,
547         ↳ sequenceToItsLocalElementLevelsConverter);
548
549     var compressed1 = new ulong[arrays.Length];
550     var compressed2 = new ulong[arrays.Length];
551     var compressed3 = new ulong[arrays.Length];
552
553     var START = 0;
554     var END = arrays.Length;
555
556     //for (int i = START; i < END; i++)
557     //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
558
559     var initialCount1 = scope2.Links.Unsync.Count();
560
561     var sw1 = Stopwatch.StartNew();
562
563     for (int i = START; i < END; i++)
564     {
565         linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
566         compressed1[i] = compressor1.Convert(arrays[i]);
567     }
568
569     var elapsed1 = sw1.Elapsed;
570
571     var balancedVariantConverter2 = new
572         ↳ BalancedVariantConverter<ulong>(scope2.Links.Unsync);
573
574     var initialCount2 = scope2.Links.Unsync.Count();
575
576     var sw2 = Stopwatch.StartNew();
577
578     for (int i = START; i < END; i++)
579     {

```

```

562         compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
563     }
564
565     var elapsed2 = sw2.Elapsed;
566
567     for (int i = START; i < END; i++)
568     {
569         linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
570     }
571
572     var initialCount3 = scope3.Links.Unsync.Count();
573
574     var sw3 = Stopwatch.StartNew();
575
576     for (int i = START; i < END; i++)
577     {
578         //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
579         compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
580     }
581
582     var elapsed3 = sw3.Elapsed;
583
584     Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
585         ↳ Optimal variant: {elapsed3}");
586
587     // Assert.True(elapsed1 > elapsed2);
588
589     // Checks
590     for (int i = START; i < END; i++)
591     {
592         var sequence1 = compressed1[i];
593         var sequence2 = compressed2[i];
594         var sequence3 = compressed3[i];
595
596         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
597             ↳ scope1.Links.Unsync);
598
599         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
600             ↳ scope2.Links.Unsync);
601
602         var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
603             ↳ scope3.Links.Unsync);
604
605         var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
606             ↳ link.IsPartialPoint());
607         var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
608             ↳ link.IsPartialPoint());
609         var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
610             ↳ link.IsPartialPoint());
611
612         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
613             ↳ arrays[i].Length > 3)
614         //    Assert.False(structure1 == structure2);
615         //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
616             ↳ arrays[i].Length > 3)
617         //    Assert.False(structure3 == structure2);
618
619         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
620         Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
621     }
622
623     Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
624         ↳ totalCharacters);
625     Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
626         ↳ totalCharacters);
627     Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
628         ↳ totalCharacters);
629
630     Console.WriteLine($"{{(double)(scope1.Links.Unsync.Count() - initialCount1) /
631         ↳ totalCharacters}} | {{(double)(scope2.Links.Unsync.Count() - initialCount2) /
632         ↳ totalCharacters}} | {{(double)(scope3.Links.Unsync.Count() - initialCount3) /
633         ↳ totalCharacters}}");
634
635     Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
636         ↳ scope2.Links.Unsync.Count() - initialCount2);
637     Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
638         ↳ scope2.Links.Unsync.Count() - initialCount2);

```

```

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($"Compressor: {elapsed1}, Balanced sequence creator:
736     ↪ {elapsed2}");
737
738     Assert.True(elapsed1 > elapsed2);
739
740     // Checks
741     for (int i = START; i < END; i++)
742     {
743         var sequence1 = compressed1[i];
744         var sequence2 = compressed2[i];
745
746         if (sequence1 != _constants.Null && sequence2 != _constants.Null)
747         {
748             var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
749             ↪ scope1.Links);
750
751             var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
752             ↪ scope2.Links);
753
754             //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
755             ↪ link.IsPartialPoint());
756             //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
757             ↪ link.IsPartialPoint());
758
759             //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
760             ↪ arrays[i].Length > 3)
761             //    Assert.False(structure1 == structure2);
762
763             Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
764         }
765     }
766
767     Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
768     Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
769
770     Debug.WriteLine($"{{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
771     ↪ totalCharacters}} | {{(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
772     ↪ totalCharacters}}");
773
774     Assert.True(scope1.Links.Count() <= scope2.Links.Count());
775
776     //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     for (ulong i = 0; i < N; i++)
788     {
789         strings.Add(RandomHelpers.Default.NextUInt64().ToString());
790     }
791
792     strings = strings.Distinct().ToList();
793
794     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
795     var totalCharacters = arrays.Select(x => x.Length).Sum();
796
797     using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
798         ↪ SequencesOptions<ulong> { UseCompression = true,
799         ↪ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
800     using (var scope2 = new TempLinksTestScope(useSequences: true))
801     {
802         scope1.Links.UseUnicode();
803         scope2.Links.UseUnicode();
804
805         var compressor1 = scope1.Sequences;
806         var compressor2 = scope2.Sequences;
807
808         var compressed1 = new ulong[arrays.Length];
809         var compressed2 = new ulong[arrays.Length];
810
811         var sw1 = Stopwatch.StartNew();
812
813         var START = 0;
814         var END = arrays.Length;
815
816         for (int i = START; i < END; i++)
817         {
818             compressed1[i] = compressor1.Create(arrays[i].ConvertToRestrictionsValues());
819         }
820
821         var elapsed1 = sw1.Elapsed;
822
823         var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
824         var sw2 = Stopwatch.StartNew();
825
826         for (int i = START; i < END; i++)
827         {
828             compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
829         }
830
831         var elapsed2 = sw2.Elapsed;
832
833         Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
834             ↪ {elapsed2}");
835
836         Assert.True(elapsed1 > elapsed2);
837
838         // Checks
839         for (int i = START; i < END; i++)
840         {
841             var sequence1 = compressed1[i];
842             var sequence2 = compressed2[i];
843
844             if (sequence1 != _constants.Null && sequence2 != _constants.Null)
845             {

```

```

844         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
845             ↪ scope1.Links);
846         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
847             ↪ scope2.Links);
848         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
849     }
850 }
851
852 Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
853 Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
854
855 Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
856     ↪ totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
857     ↪ totalCharacters}");
858
859 // Can be worse than balanced variant
860 //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
861 //compressor1.ValidateFrequencies();
862 }
863
864 [Fact]
865 public static void AllTreeBreakDownAtSequencesCreationBugTest()
866 {
867     // Made out of AllPossibleConnectionsTest test.
868
869     //const long sequenceLength = 5; //100% bug
870     const long sequenceLength = 4; //100% bug
871     //const long sequenceLength = 3; //100% _no_bug (ok)
872
873     using (var scope = new TempLinksTestScope(useSequences: true))
874     {
875         var links = scope.Links;
876         var sequences = scope.Sequences;
877
878         var sequence = new ulong[sequenceLength];
879         for (var i = 0; i < sequenceLength; i++)
880         {
881             sequence[i] = links.Create();
882         }
883
884         var createResults = sequences.CreateAllVariants2(sequence);
885         Global.Trash = createResults;
886
887         for (var i = 0; i < sequenceLength; i++)
888         {
889             links.Delete(sequence[i]);
890         }
891     }
892 }
893
894 [Fact]
895 public static void AllPossibleConnectionsTest()
896 {
897     const long sequenceLength = 5;
898
899     using (var scope = new TempLinksTestScope(useSequences: true))
900     {
901         var links = scope.Links;
902         var sequences = scope.Sequences;
903
904         var sequence = new ulong[sequenceLength];
905         for (var i = 0; i < sequenceLength; i++)
906         {
907             sequence[i] = links.Create();
908         }
909
910         var createResults = sequences.CreateAllVariants2(sequence);
911         var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
912
913         for (var i = 0; i < 1; i++)
914         {
915             var sw1 = Stopwatch.StartNew();
916             var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
917
918

```

```

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

```

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

```
1 using System.IO;
2 using Platform.Disposables;
3 using Platform.Data.Doublets.ResizableDirectMemory;
4 using Platform.Data.Doublets.Sequences;
5 using Platform.Data.Doublets.Decorators;
6
7 namespace Platform.Data.Doublets.Tests
8 {
9     public class TempLinksTestScope : DisposableBase
10    {
11        public ILinks<ulong> MemoryAdapter { get; }
12        public SynchronizedLinks<ulong> Links { get; }
13        public Sequences.Sequences Sequences { get; }
14        public string TempFilename { get; }
15        public string TempTransactionLogFilename { get; }
16        private readonly bool _deleteFiles;
17
18        public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool
        ↪ 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    }
22 }
```

```

21     var setter = new Setter<T>(constants.Null);
22     links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
23
24     Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
25
26     var linkAddress = links.Create();
27
28     var link = new Link<T>(links.GetLink(linkAddress));
29
30     Assert.True(link.Count == 3);
31     Assert.True(equalityComparer.Equals(link.Index, linkAddress));
32     Assert.True(equalityComparer.Equals(link.Source, constants.Null));
33     Assert.True(equalityComparer.Equals(link.Target, constants.Null));
34
35     Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.One));
36
37     // Get first link
38     setter = new Setter<T>(constants.Null);
39     links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);
40
41     Assert.True(equalityComparer.Equals(setter.Result, linkAddress));
42
43     // Update link to reference itself
44     links.Update(linkAddress, linkAddress, linkAddress);
45
46     link = new Link<T>(links.GetLink(linkAddress));
47
48     Assert.True(equalityComparer.Equals(link.Source, linkAddress));
49     Assert.True(equalityComparer.Equals(link.Target, linkAddress));
50
51     // Update link to reference null (prepare for delete)
52     var updated = links.Update(linkAddress, constants.Null, constants.Null);
53
54     Assert.True(equalityComparer.Equals(updated, linkAddress));
55
56     link = new Link<T>(links.GetLink(linkAddress));
57
58     Assert.True(equalityComparer.Equals(link.Source, constants.Null));
59     Assert.True(equalityComparer.Equals(link.Target, constants.Null));
60
61     // Delete link
62     links.Delete(linkAddress);
63
64     Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));
65
66     setter = new Setter<T>(constants.Null);
67     links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
68
69     Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
70 }
71
72 public static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
73 {
74     // Constants
75     var constants = links.Constants;
76     var equalityComparer = EqualityComparer<T>.Default;
77
78     var h106E = new Hybrid<T>(106L, isExternal: true);
79     var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
80     var h108E = new Hybrid<T>(-108L);
81
82     Assert.Equal(106L, h106E.AbsoluteValue);
83     Assert.Equal(107L, h107E.AbsoluteValue);
84     Assert.Equal(108L, h108E.AbsoluteValue);
85
86     // Create Link (External -> External)
87     var linkAddress1 = links.Create();
88
89     links.Update(linkAddress1, h106E, h108E);
90
91     var link1 = new Link<T>(links.GetLink(linkAddress1));
92
93     Assert.True(equalityComparer.Equals(link1.Source, h106E));
94     Assert.True(equalityComparer.Equals(link1.Target, h108E));
95
96     // Create Link (Internal -> External)
97     var linkAddress2 = links.Create();
98
99     links.Update(linkAddress2, linkAddress1, h108E);
100

```

```

101     var link2 = new Link<T>(links.GetLink(linkAddress2));
102
103     Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
104     Assert.True(equalityComparer.Equals(link2.Target, h108E));
105
106     // Create Link (Internal -> Internal)
107     var linkAddress3 = links.Create();
108
109     links.Update(linkAddress3, linkAddress1, linkAddress2);
110
111     var link3 = new Link<T>(links.GetLink(linkAddress3));
112
113     Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
114     Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));
115
116     // Search for created link
117     var setter1 = new Setter<T>(constants.Null);
118     links.Each(h106E, h108E, setter1.SetAndReturnFalse);
119
120     Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));
121
122     // Search for nonexistent link
123     var setter2 = new Setter<T>(constants.Null);
124     links.Each(h106E, h107E, setter2.SetAndReturnFalse);
125
126     Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));
127
128     // Update link to reference null (prepare for delete)
129     var updated = links.Update(linkAddress3, constants.Null, constants.Null);
130
131     Assert.True(equalityComparer.Equals(updated, linkAddress3));
132
133     link3 = new Link<T>(links.GetLink(linkAddress3));
134
135     Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
136     Assert.True(equalityComparer.Equals(link3.Target, constants.Null));
137
138     // Delete link
139     links.Delete(linkAddress3);
140
141     Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Two));
142
143     var setter3 = new Setter<T>(constants.Null);
144     links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);
145
146     Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
147 }
148
149 public static void TestMultipleRandomCreationsAndDeletions<TLink>(this ILinks<TLink>
→ links, int maximumOperationsPerCycle)
150 {
151     var comparer = Comparer<TLink>.Default;
152     for (var N = 1; N < maximumOperationsPerCycle; N++)
153     {
154         var random = new System.Random(N);
155         var created = 0;
156         var deleted = 0;
157         for (var i = 0; i < N; i++)
158         {
159             long linksCount = (Integer<TLink>)links.Count();
160             var createPoint = random.NextBoolean();
161             if (linksCount > 2 && createPoint)
162             {
163                 var linksAddressRange = new Range<ulong>(1, (ulong)linksCount);
164                 TLink source = (Integer<TLink>)random.NextUInt64(linksAddressRange);
165                 TLink target = (Integer<TLink>)random.NextUInt64(linksAddressRange);
166                 → //-V3086
167                 var resultLink = links.CreateAndUpdate(source, target);
168                 if (comparer.Compare(resultLink, (Integer<TLink>)linksCount) > 0)
169                 {
170                     created++;
171                 }
172             }
173             else
174             {
175                 links.Create();
176                 created++;
177             }
178         }
179     }

```

```

178         Assert.True(created == (Integer<TLink>)links.Count());
179         for (var i = 0; i < N; i++)
180         {
181             TLink link = (Integer<TLink>)(i + 1);
182             if (links.Exists(link))
183             {
184                 links.Delete(link);
185                 deleted++;
186             }
187         }
188         Assert.True((Integer<TLink>)links.Count() == 0);
189     }
190 }
191 }
192 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.IO;
5  using System.Text;
6  using System.Threading;
7  using System.Threading.Tasks;
8  using Xunit;
9  using Platform.Disposables;
10 using Platform.IO;
11 using Platform.Ranges;
12 using Platform.Random;
13 using Platform.Timestamps;
14 using Platform.Singletons;
15 using Platform.Counters;
16 using Platform.Diagnostics;
17 using Platform.Data.Doublets.ResizableDirectMemory;
18 using Platform.Data.Doublets.Decorators;
19
20 namespace Platform.Data.Doublets.Tests
21 {
22     public static class UInt64LinksTests
23     {
24         private static readonly LinksConstants<ulong> _constants =
25             ↪ Default<LinksConstants<ulong>>.Instance;
26
27         private const long Iterations = 10 * 1024;
28
29         #region Concept
30
31         [Fact]
32         public static void MultipleCreateAndDeleteTest()
33         {
34             using (var scope = new TempLinksTestScope())
35             {
36                 scope.Links.TestMultipleRandomCreationsAndDeletions(100);
37             }
38
39             [Fact]
40             public static void CascadeUpdateTest()
41             {
42                 var itself = _constants.Itself;
43
44                 using (var scope = new TempLinksTestScope(useLog: true))
45                 {
46                     var links = scope.Links;
47
48                     var l1 = links.Create();
49                     var l2 = links.Create();
50
51                     l2 = links.Update(l2, l2, l1, l2);
52
53                     links.CreateAndUpdate(l2, itself);
54                     links.CreateAndUpdate(l2, itself);
55
56                     l2 = links.Update(l2, l1);
57
58                     links.Delete(l2);
59
60                     Global.Trash = links.Count();
61
62                     links.Unsync.DisposeIfPossible(); // Close links to access log
63

```

```

64         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope
        ↪ e.TempTransactionLogFilename);
65     }
66 }
67
68 [Fact]
69 public static void BasicTransactionLogTest()
70 {
71     using (var scope = new TempLinksTestScope(useLog: true))
72     {
73         var links = scope.Links;
74         var l1 = links.Create();
75         var l2 = links.Create();
76
77         Global.Trash = links.Update(l2, l2, l1, l2);
78
79         links.Delete(l1);
80
81         links.Unsync.DisposeIfPossible(); // Close links to access log
82
83         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope
        ↪ e.TempTransactionLogFilename);
84     }
85 }
86
87 [Fact]
88 public static void TransactionAutoRevertedTest()
89 {
90     // Auto Reverted (Because no commit at transaction)
91     using (var scope = new TempLinksTestScope(useLog: true))
92     {
93         var links = scope.Links;
94         var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
95         using (var transaction = transactionsLayer.BeginTransaction())
96         {
97             var l1 = links.Create();
98             var l2 = links.Create();
99
100             links.Update(l2, l2, l1, l2);
101         }
102
103         Assert.Equal(0UL, links.Count());
104
105         links.Unsync.DisposeIfPossible();
106
107         var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s
        ↪ cope.TempTransactionLogFilename);
108         Assert.Single(transitions);
109     }
110 }
111
112 [Fact]
113 public static void TransactionUserCodeErrorNoDataSavedTest()
114 {
115     // User Code Error (Autoreverted), no data saved
116     var itself = _constants.Itself;
117
118     TempLinksTestScope lastScope = null;
119     try
120     {
121         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
        ↪ useLog: true))
122         {
123             var links = scope.Links;
124             var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor
        ↪ atorBase<ulong>)links.Unsync).Links;
125             using (var transaction = transactionsLayer.BeginTransaction())
126             {
127                 var l1 = links.CreateAndUpdate(itself, itself);
128                 var l2 = links.CreateAndUpdate(itself, itself);
129
130                 l2 = links.Update(l2, l2, l1, l2);
131
132                 links.CreateAndUpdate(l2, itself);
133                 links.CreateAndUpdate(l2, itself);
134
135                 //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi
        ↪ tion>(scope.TempTransactionLogFilename);
136

```



```

137         l2 = links.Update(l2, l1);
138
139         links.Delete(l2);
140
141         ExceptionThrower();
142
143         transaction.Commit();
144     }
145
146     Global.Trash = links.Count();
147 }
148
149 catch
150 {
151     Assert.False(lastScope == null);
152
153     var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l
        ↳ astScope.TempTransactionLogFilename);
154
155     Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
        ↳ transitions[0].After.IsNull());
156
157     lastScope.DeleteFiles();
158 }
159 }
160
161 [Fact]
162 public static void TransactionUserCodeErrorSomeDataSavedTest()
163 {
164     // User Code Error (Autoreverted), some data saved
165     var itself = _constants.Itself;
166
167     TempLinksTestScope lastScope = null;
168     try
169     {
170         ulong l1;
171         ulong l2;
172
173         using (var scope = new TempLinksTestScope(useLog: true))
174         {
175             var links = scope.Links;
176             l1 = links.CreateAndUpdate(itself, itself);
177             l2 = links.CreateAndUpdate(itself, itself);
178
179             l2 = links.Update(l2, l2, l1, l2);
180
181             links.CreateAndUpdate(l2, itself);
182             links.CreateAndUpdate(l2, itself);
183
184             links.Unsync.DisposeIfPossible();
185
186             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
        ↳ scope.TempTransactionLogFilename);
187         }
188
189         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
        ↳ useLog: true))
190         {
191             var links = scope.Links;
192             var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
193             using (var transaction = transactionsLayer.BeginTransaction())
194             {
195                 l2 = links.Update(l2, l1);
196
197                 links.Delete(l2);
198
199                 ExceptionThrower();
200
201                 transaction.Commit();
202             }
203
204             Global.Trash = links.Count();
205         }
206     }
207     catch
208     {
209         Assert.False(lastScope == null);
210
211         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
        ↳ Scope.TempTransactionLogFilename);

```

```

212         lastScope.DeleteFiles();
213     }
214 }
215
216 [Fact]
217 public static void TransactionCommit()
218 {
219     var itself = _constants.Itself;
220
221     var tempDatabaseFilename = Path.GetTempFileName();
222     var tempTransactionLogFilename = Path.GetTempFileName();
223
224     // Commit
225     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
226         ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
227         ↪ tempTransactionLogFilename))
228     using (var links = new UInt64Links(memoryAdapter))
229     {
230         using (var transaction = memoryAdapter.BeginTransaction())
231         {
232             var l1 = links.CreateAndUpdate(itself, itself);
233             var l2 = links.CreateAndUpdate(itself, itself);
234
235             Global.Trash = links.Update(l2, l2, l1, l2);
236
237             links.Delete(l1);
238
239             transaction.Commit();
240         }
241
242         Global.Trash = links.Count();
243
244         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
245             ↪ sactionLogFilename);
246     }
247
248 [Fact]
249 public static void TransactionDamage()
250 {
251     var itself = _constants.Itself;
252
253     var tempDatabaseFilename = Path.GetTempFileName();
254     var tempTransactionLogFilename = Path.GetTempFileName();
255
256     // Commit
257     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
258         ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
259         ↪ 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         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
277             ↪ sactionLogFilename);
278
279         // Damage database
280
281         FileHelpers.WriteFirst(tempTransactionLogFilename, new
282             ↪ UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
283
284         // Try load damaged database
285         try
286         {
287             // TODO: Fix

```

```

284         using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
        ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
        ↪ tempTransactionLogFilename))
285     using (var links = new UInt64Links(memoryAdapter))
286     {
287         Global.Trash = links.Count();
288     }
289 }
290 catch (NotSupportedException ex)
291 {
292     Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
        ↪ yet.");
293 }
294
295 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
    ↪ sactionLogFilename);
296
297 File.Delete(tempDatabaseFilename);
298 File.Delete(tempTransactionLogFilename);
299 }
300
301 [Fact]
302 public static void Bug1Test()
303 {
304     var tempDatabaseFilename = Path.GetTempFileName();
305     var tempTransactionLogFilename = Path.GetTempFileName();
306
307     var itself = _constants.Itself;
308
309     // User Code Error (Autoreverted), some data saved
310     try
311     {
312         ulong l1;
313         ulong l2;
314
315         using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
        ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
        ↪ tempTransactionLogFilename))
316     using (var links = new UInt64Links(memoryAdapter))
317     {
318         l1 = links.CreateAndUpdate(itself, itself);
319         l2 = links.CreateAndUpdate(itself, itself);
320
321         l2 = links.Update(l2, l2, l1, l2);
322
323         links.CreateAndUpdate(l2, itself);
324         links.CreateAndUpdate(l2, itself);
325     }
326
327     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
    ↪ TransactionLogFilename);
328
329     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
        ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
        ↪ tempTransactionLogFilename))
330     using (var links = new UInt64Links(memoryAdapter))
331     {
332         using (var transaction = memoryAdapter.BeginTransaction())
333         {
334             l2 = links.Update(l2, l1);
335
336             links.Delete(l2);
337
338             ExceptionThrower();
339
340             transaction.Commit();
341         }
342
343         Global.Trash = links.Count();
344     }
345 }
346 catch
347 {
348     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
    ↪ TransactionLogFilename);
349 }
350
351 File.Delete(tempDatabaseFilename);

```

```

352     File.Delete(tempTransactionLogFilename);
353 }
354
355 private static void ExceptionThrower() => throw new InvalidOperationException();
356
357 [Fact]
358 public static void PathsTest()
359 {
360     var source = _constants.SourcePart;
361     var target = _constants.TargetPart;
362
363     using (var scope = new TempLinksTestScope())
364     {
365         var links = scope.Links;
366         var l1 = links.CreatePoint();
367         var l2 = links.CreatePoint();
368
369         var r1 = links.GetByKeys(l1, source, target, source);
370         var r2 = links.CheckPathExistence(l2, l2, l2, l2);
371     }
372 }
373
374 [Fact]
375 public static void RecursiveStringFormattingTest()
376 {
377     using (var scope = new TempLinksTestScope(useSequences: true))
378     {
379         var links = scope.Links;
380         var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.
381
382         var a = links.CreatePoint();
383         var b = links.CreatePoint();
384         var c = links.CreatePoint();
385
386         var ab = links.CreateAndUpdate(a, b);
387         var cb = links.CreateAndUpdate(c, b);
388         var ac = links.CreateAndUpdate(a, c);
389
390         a = links.Update(a, c, b);
391         b = links.Update(b, a, c);
392         c = links.Update(c, a, b);
393
394         Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
395         Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
396         Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
397
398         Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
399             ↳ "(5:(4:5 (6:5 4)) 6)");
400         Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
401             ↳ "(6:(5:(4:5 6) 6) 4)");
402         Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
403             ↳ "(4:(5:4 (6:5 4)) 6)");
404
405         // TODO: Think how to build balanced syntax tree while formatting structure (eg.
406         ↳ "(4:(5:4 6) (6:5 4))" instead of "(4:(5:4 (6:5 4)) 6)"
407
408         Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
409             ↳ "{5}{5}{4}{6}");
410         Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
411             ↳ "{5}{6}{6}{4}");
412         Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
413             ↳ "{4}{5}{4}{6}");
414     }
415 }
416
417 private static void DefaultFormatter(StringBuilder sb, ulong link)
418 {
419     sb.Append(link.ToString());
420 }
421
422 #endregion
423
424 #region Performance
425
426 /*
427 public static void RunAllPerformanceTests()
428 {
429     try
430     {

```

```

424         links.TestLinksInSteps();
425     }
426     catch (Exception ex)
427     {
428         ex.WriteToConsole();
429     }
430
431     return;
432
433     try
434     {
435         //ThreadPool.SetMaxThreads(2, 2);
436
437         // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
↪ результат
438         // Также это дополнительно помогает в отладке
439         // Увеличивает вероятность попадания информации в кэши
440         for (var i = 0; i < 10; i++)
441         {
442             //0 - 10 ГБ
443             //Каждые 100 МБ срез цифр
444
445             //links.TestGetSourceFunction();
446             //links.TestGetSourceFunctionInParallel();
447             //links.TestGetTargetFunction();
448             //links.TestGetTargetFunctionInParallel();
449             links.Create64BillionLinks();
450
451             links.TestRandomSearchFixed();
452             //links.Create64BillionLinksInParallel();
453             links.TestEachFunction();
454             //links.TestForeach();
455             //links.TestParallelForeach();
456         }
457
458         links.TestDeletionOfAllLinks();
459     }
460     catch (Exception ex)
461     {
462         ex.WriteToConsole();
463     }
464 }*/
465
466 /*
467 public static void TestLinksInSteps()
468 {
469     const long gibibyte = 1024 * 1024 * 1024;
470     const long mebibyte = 1024 * 1024;
471
472     var totalLinksToCreate = gibibyte /
↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
473     var linksStep = 102 * mebibyte /
↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
474
475     var creationMeasurements = new List<TimeSpan>();
476     var searchMeasurements = new List<TimeSpan>();
477     var deletionMeasurements = new List<TimeSpan>();
478
479     GetBaseRandomLoopOverhead(linksStep);
480     GetBaseRandomLoopOverhead(linksStep);
481
482     var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
483
484     ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
485
486     var loops = totalLinksToCreate / linksStep;
487
488     for (int i = 0; i < loops; i++)
489     {
490         creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
491         searchMeasurements.Add(Measure(() => links.RunRandomSearches(linksStep)));
492
493         Console.WriteLine("\rC + S {0}/{1}", i + 1, loops);
494     }
495
496     ConsoleHelpers.Debug();
497
498     for (int i = 0; i < loops; i++)
499     {
500

```

```

501         deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
502
503         Console.WriteLine("\rD {0}/{1}", i + 1, loops);
504     }
505
506     ConsoleHelpers.Debug();
507
508     ConsoleHelpers.Debug("C S D");
509
510     for (int i = 0; i < loops; i++)
511     {
512         ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
↵ searchMeasurements[i], deletionMeasurements[i]);
513     }
514
515     ConsoleHelpers.Debug("C S D (no overhead)");
516
517     for (int i = 0; i < loops; i++)
518     {
519         ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] - stepLoopOverhead,
↵ searchMeasurements[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
520     }
521
522     ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
↵ links.Total);
523 }
524
525 private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
↵ amountToCreate)
526 {
527     for (long i = 0; i < amountToCreate; i++)
528         links.Create(0, 0);
529 }
530
531 private static TimeSpan GetBaseRandomLoopOverhead(long loops)
532 {
533     return Measure(() =>
534     {
535         ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
536         ulong result = 0;
537         for (long i = 0; i < loops; i++)
538         {
539             var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
540             var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
541
542             result += maxValue + source + target;
543         }
544         Global.Trash = result;
545     });
546 }
547
548
549 [Fact(Skip = "performance test")]
550 public static void GetSourceTest()
551 {
552     using (var scope = new TempLinksTestScope())
553     {
554         var links = scope.Links;
555         ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
↵ Iterations);
556
557         ulong counter = 0;
558
559         //var firstLink = links.First();
560         // Создаём одну связь, из которой будет производить считывание
561         var firstLink = links.Create();
562
563         var sw = Stopwatch.StartNew();
564
565         // Тестируем саму функцию
566         for (ulong i = 0; i < Iterations; i++)
567         {
568             counter += links.GetSource(firstLink);
569         }
570
571         var elapsedTime = sw.Elapsed;
572
573         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
574

```

```

575 // Удаляем связь, из которой производилось считывание
576 links.Delete(firstLink);
577
578 ConsoleHelpers.Debug(
579     "{0} Iterations of GetSource function done in {1} ({2} Iterations per
    ↳ second), counter result: {3}",
    Iterations, elapsedTime, (long)iterationsPerSecond, counter);
580 }
581 }
582 }
583
584 [Fact(Skip = "performance test")]
585 public static void GetSourceInParallel()
586 {
587     using (var scope = new TempLinksTestScope())
588     {
589         var links = scope.Links;
590         ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
    ↳ parallel.", Iterations);
591
592         long counter = 0;
593
594         //var firstLink = links.First();
595         var firstLink = links.Create();
596
597         var sw = Stopwatch.StartNew();
598
599         // Тестируем саму функцию
600         Parallel.For(0, Iterations, x =>
601         {
602             Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
603             //Interlocked.Increment(ref counter);
604         });
605
606         var elapsedTime = sw.Elapsed;
607
608         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
609
610         links.Delete(firstLink);
611
612         ConsoleHelpers.Debug(
613             "{0} Iterations of GetSource function done in {1} ({2} Iterations per
    ↳ second), counter result: {3}",
            Iterations, elapsedTime, (long)iterationsPerSecond, counter);
614     }
615 }
616
617 [Fact(Skip = "performance test")]
618 public static void TestGetTarget()
619 {
620     using (var scope = new TempLinksTestScope())
621     {
622         var links = scope.Links;
623         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
    ↳ Iterations);
624
625         ulong counter = 0;
626
627         //var firstLink = links.First();
628         var firstLink = links.Create();
629
630         var sw = Stopwatch.StartNew();
631
632         for (ulong i = 0; i < Iterations; i++)
633         {
634             counter += links.GetTarget(firstLink);
635         }
636
637         var elapsedTime = sw.Elapsed;
638
639         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
640
641         links.Delete(firstLink);
642
643         ConsoleHelpers.Debug(
644             "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
    ↳ second), counter result: {3}",
            Iterations, elapsedTime, (long)iterationsPerSecond, counter);
645     }
646 }
647 }
648

```

```

649 [Fact(Skip = "performance test")]
650 public static void TestGetTargetInParallel()
651 {
652     using (var scope = new TempLinksTestScope())
653     {
654         var links = scope.Links;
655         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
656             ↳ parallel.", Iterations);
657
658         long counter = 0;
659
660         //var firstLink = links.First();
661         var firstLink = links.Create();
662
663         var sw = Stopwatch.StartNew();
664
665         Parallel.For(0, Iterations, x =>
666         {
667             Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
668             //Interlocked.Increment(ref counter);
669         });
670
671         var elapsedTime = sw.Elapsed;
672
673         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
674
675         links.Delete(firstLink);
676
677         ConsoleHelpers.Debug(
678             "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
679             ↳ second), counter result: {3}",
680             Iterations, elapsedTime, (long)iterationsPerSecond, counter);
681     }
682
683     // TODO: Заполнить базу данных перед тестом
684     /*
685     [Fact]
686     public void TestRandomSearchFixed()
687     {
688         var tempFilename = Path.GetTempFileName();
689
690         using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
691             ↳ DefaultLinksSizeStep))
692         {
693             long iterations = 64 * 1024 * 1024 /
694             ↳ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
695
696             ulong counter = 0;
697             var maxLink = links.Total;
698
699             ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);
700
701             var sw = Stopwatch.StartNew();
702
703             for (var i = iterations; i > 0; i--)
704             {
705                 var source =
706                 ↳ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
707                 var target =
708                 ↳ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
709
710                 counter += links.Search(source, target);
711             }
712
713             var elapsedTime = sw.Elapsed;
714
715             var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
716
717             ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
718             ↳ Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
719             ↳ counter);
720         }
721
722         File.Delete(tempFilename);
723     }*/
724
725     [Fact(Skip = "useless: 0(0), was dependent on creation tests")]

```



```

720 public static void TestRandomSearchAll()
721 {
722     using (var scope = new TempLinksTestScope())
723     {
724         var links = scope.Links;
725         ulong counter = 0;
726
727         var maxLink = links.Count();
728
729         var iterations = links.Count();
730
731         ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
732             ↪ links.Count());
733
734         var sw = Stopwatch.StartNew();
735
736         for (var i = iterations; i > 0; i--)
737         {
738             var linksAddressRange = new
739                 ↪ Range<ulong>(_constants.PossibleInnerReferencesRange.Minimum, maxLink);
740
741             var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
742             var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
743
744             counter += links.SearchOrDefault(source, target);
745         }
746
747         var elapsedTime = sw.Elapsed;
748
749         var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
750
751         ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
752             ↪ Iterations per second), c: {3}",
753             iterations, elapsedTime, (long)iterationsPerSecond, counter);
754     }
755 }
756
757 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
758 public static void TestEach()
759 {
760     using (var scope = new TempLinksTestScope())
761     {
762         var links = scope.Links;
763
764         var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
765
766         ConsoleHelpers.Debug("Testing Each function.");
767
768         var sw = Stopwatch.StartNew();
769
770         links.Each(counter.IncrementAndReturnTrue);
771
772         var elapsedTime = sw.Elapsed;
773
774         var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
775
776         ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
777             ↪ links per second)",
778             counter, elapsedTime, (long)linksPerSecond);
779     }
780 }
781
782 /*
783 [Fact]
784 public static void TestForeach()
785 {
786     var tempFilename = Path.GetTempFileName();
787
788     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
789         ↪ DefaultLinksSizeStep))
790     {
791         ulong counter = 0;
792
793         ConsoleHelpers.Debug("Testing foreach through links.");
794
795         var sw = Stopwatch.StartNew();
796
797         //foreach (var link in links)
798         //{
799             counter++;
800         }
801     }
802 }

```

```

795         //}
796
797         var elapsedTime = sw.Elapsed;
798
799         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
800
801         ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
↪ links per second)", counter, elapsedTime, (long)linksPerSecond);
802     }
803
804     File.Delete(tempFilename);
805 }
806 */
807
808 /*
809 [Fact]
810 public static void TestParallelForeach()
811 {
812     var tempFilename = Path.GetTempFileName();
813
814     using (var links = new Platform.Links.Data.Core.Doublents.Links(tempFilename,
↪ DefaultLinksSizeStep))
815     {
816         long counter = 0;
817
818         ConsoleHelpers.Debug("Testing parallel foreach through links.");
819
820         var sw = Stopwatch.StartNew();
821
822         //Parallel.ForEach((IEnumerable<ulong>)links, x =>
823         //{
824             //    Interlocked.Increment(ref counter);
825         //});
826
827         var elapsedTime = sw.Elapsed;
828
829         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
830
831         ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
↪ {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
832     }
833
834     File.Delete(tempFilename);
835 }
836 */
837
838 [Fact(Skip = "performance test")]
839 public static void Create64BillionLinks()
840 {
841     using (var scope = new TempLinksTestScope())
842     {
843         var links = scope.Links;
844         var linksBeforeTest = links.Count();
845
846         long linksToCreate = 64 * 1024 * 1024 /
↪ UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
847
848         ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
849
850         var elapsedTime = Performance.Measure(() =>
851         {
852             for (long i = 0; i < linksToCreate; i++)
853             {
854                 links.Create();
855             }
856         });
857
858         var linksCreated = links.Count() - linksBeforeTest;
859         var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
860
861         ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
862
863         ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
↪ linksCreated, elapsedTime,
864             (long)linksPerSecond);
865     }
866 }
867
868 [Fact(Skip = "performance test")]

```

```

public static void Create64BillionLinksInParallel()
{
    using (var scope = new TempLinksTestScope())
    {
        var links = scope.Links;
        var linksBeforeTest = links.Count();

        var sw = Stopwatch.StartNew();

        long linksToCreate = 64 * 1024 * 1024 /
            ↳ UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;

        ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);
        Parallel.For(0, linksToCreate, x => links.Create());

        var elapsedTime = sw.Elapsed;

        var linksCreated = links.Count() - linksBeforeTest;
        var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;

        ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
            ↳ linksCreated, elapsedTime,
            (long)linksPerSecond);
    }
}

[Fact(Skip = "useless: 0(0), was dependent on creation tests")]
public static void TestDeletionOfAllLinks()
{
    using (var scope = new TempLinksTestScope())
    {
        var links = scope.Links;
        var linksBeforeTest = links.Count();

        ConsoleHelpers.Debug("Deleting all links");

        var elapsedTime = Performance.Measure(links.DeleteAll);

        var linksDeleted = linksBeforeTest - links.Count();
        var linksPerSecond = linksDeleted / elapsedTime.TotalSeconds;

        ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
            ↳ linksDeleted, elapsedTime,
            (long)linksPerSecond);
    }
}

#endregion

```

```
./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                 }
29             }
30         }
31     }
32 }

```

```

26         unaryNumbers[i] = toUnaryNumberConverter.Convert(numbers[i]);
27     }
28     var fromUnaryNumberConverterUsingOrOperation = new
    ↪     UnaryNumberToAddressOrOperationConverter<ulong>(links,
    ↪     powerOf2ToUnaryNumberConverter);
29     var fromUnaryNumberConverterUsingAddOperation = new
    ↪     UnaryNumberToAddressAddOperationConverter<ulong>(links, one);
30     for (int i = 0; i < N; i++)
31     {
32         Assert.Equal(numbers[i],
    ↪         fromUnaryNumberConverterUsingOrOperation.Convert(unaryNumbers[i]));
33         Assert.Equal(numbers[i],
    ↪         fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));
34     }
35 }
36 }
37 }
38 }

```

./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
    ↪                 PowerOf2ToUnaryNumberConverter<ulong>(links, one);
29                 var addressToUnaryNumberConverter = new
    ↪                 AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
30                 var unaryNumberToAddressConverter = new
    ↪                 UnaryNumberToAddressOrOperationConverter<ulong>(links,
    ↪                 powerOf2ToUnaryNumberConverter);
31                 TestCharAndUnicodeSymbolConverters(links, meaningRoot,
    ↪                 addressToUnaryNumberConverter, unaryNumberToAddressConverter);
32             }
33         }
34
35         [Fact]
36         public static void CharAndRawNumberUnicodeSymbolConvertersTest()
37         {
38             using (var scope = new Scope<Types<HeapResizableDirectMemory,
    ↪             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,
    ↪                 addressToRawNumberConverter, rawNumberToAddressConverter);
45             }
46         }
47
48         private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
    ↪         meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
    ↪         numberToAddressConverter)
49         {
50             var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);

```

```

51     var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links,
    ↪ addressToNumberConverter, unicodeSymbolMarker);
52     var originalCharacter = 'H';
53     var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);
54     var unicodeSymbolCriterionMatcher = new UnicodeSymbolCriterionMatcher<ulong>(links,
    ↪ unicodeSymbolMarker);
55     var unicodeSymbolToCharConverter = new UnicodeSymbolToCharConverter<ulong>(links,
    ↪ numberToAddressConverter, unicodeSymbolCriterionMatcher);
56     var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
57     Assert.Equal(originalCharacter, resultingCharacter);
58 }
59
60 [Fact]
61 public static void StringAndUnicodeSequenceConvertersTest()
62 {
63     using (var scope = new TempLinksTestScope())
64     {
65         var links = scope.Links;
66
67         var itself = links.Constants.Itself;
68
69         var meaningRoot = links.CreatePoint();
70         var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
71         var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
72         var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
73         var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
74         var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);
75
76         var powerOf2ToUnaryNumberConverter = new
    ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
77         var addressToUnaryNumberConverter = new
    ↪ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
78         var charToUnicodeSymbolConverter = new
    ↪ CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
    ↪ unicodeSymbolMarker);
79
80         var unaryNumberToAddressConverter = new
    ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links,
    ↪ powerOf2ToUnaryNumberConverter);
81         var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
82         var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
    ↪ frequencyMarker, unaryOne, unaryNumberIncrementer);
83         var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
    ↪ frequencyPropertyMarker, frequencyMarker);
84         var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
    ↪ frequencyPropertyOperator, frequencyIncrementer);
85         var linkToItsFrequencyNumberConverter = new
    ↪ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
    ↪ unaryNumberToAddressConverter);
86         var sequenceToItsLocalElementLevelsConverter = new
    ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
    ↪ linkToItsFrequencyNumberConverter);
87         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
    ↪ sequenceToItsLocalElementLevelsConverter);
88
89         var stringToUnicodeSequenceConverter = new
    ↪ StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
    ↪ index, optimalVariantConverter, unicodeSequenceMarker);
90
91         var originalString = "Hello";
92
93         var unicodeSequenceLink =
    ↪ stringToUnicodeSequenceConverter.Convert(originalString);
94
95         var unicodeSymbolCriterionMatcher = new
    ↪ UnicodeSymbolCriterionMatcher<ulong>(links, unicodeSymbolMarker);
96         var unicodeSymbolToCharConverter = new
    ↪ UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
    ↪ unicodeSymbolCriterionMatcher);
97
98         var unicodeSequenceCriterionMatcher = new
    ↪ UnicodeSequenceCriterionMatcher<ulong>(links, unicodeSequenceMarker);
99
100        var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
    ↪ unicodeSymbolCriterionMatcher.IsMatched);

```

```
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, 146
./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, 163
./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/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, 126
./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, 137
./Platform.Data.Doublets/Unicode/UnicodeMap.cs, 138
./Platform.Data.Doublets/Unicode/UnicodeSequenceCriterionMatcher.cs, 140
./Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs, 141
./Platform.Data.Doublets/Unicode/UnicodeSymbolCriterionMatcher.cs, 141
./Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs, 141