

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  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Data.Doublets.Decorators
4  {
5      /// <remarks>
6      /// <para>Must be used in conjunction with NonNullContentsLinkDeletionResolver.</para>
7      /// <para>Должен использоваться вместе с NonNullContentsLinkDeletionResolver.</para>
8      /// </remarks>
9      public class LinksCascadeUsagesResolver<TLink> : LinksDecoratorBase<TLink>
10     {
11         public LinksCascadeUsagesResolver(ILinks<TLink> links) : base(links) { }
12
13         public override void Delete(TLink linkIndex)
14         {
15             // Use Facade (the last decorator) to ensure recursion working correctly
16             Facade.DeleteAllUsages(linkIndex);
17             Links.Delete(linkIndex);
18         }
19     }
20 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Data.Constants;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Decorators
8  {
9      public abstract class LinksDecoratorBase<TLink> : LinksOperatorBase<TLink>, ILinks<TLink>
10     {
11         public LinksCombinedConstants<TLink, TLink, int> Constants { get; }
12
13         public ILinks<TLink> _facade;
14
15         public ILinks<TLink> Facade
16         {
17             get => _facade;
18             private set
19             {
20                 _facade = value;
21                 if (Links is LinksDecoratorBase<TLink> decorator)
22                 {
23                     decorator.Facade = value;
24                 }
25             }
26         }
27
28         protected LinksDecoratorBase(ILinks<TLink> links) : base(links)
29         {
30             Constants = links.Constants;
31             Facade = this;
32         }
33
34         public virtual TLink Count(IList<TLink> restriction) => Links.Count(restriction);
35
36         public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
37             ↪ => Links.Each(handler, restrictions);

```

```

37
38     public virtual TLink Create() => Links.Create();
39
40     public virtual TLink Update(IList<TLink> restrictions) => Links.Update(restrictions);
41
42     public virtual void Delete(TLink link) => Links.Delete(link);
43 }
44 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Disposables;
4  using Platform.Data.Constants;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Decorators
9  {
10     public abstract class LinksDisposableDecoratorBase<TLink> : DisposableBase, ILinks<TLink>
11     {
12         public LinksCombinedConstants<TLink, TLink, int> Constants { get; }
13
14         public ILinks<TLink> Links { get; }
15
16         protected LinksDisposableDecoratorBase(ILinks<TLink> links)
17         {
18             Links = links;
19             Constants = links.Constants;
20         }
21
22         public virtual TLink Count(IList<TLink> restriction) => Links.Count(restriction);
23
24         public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
25             => Links.Each(handler, restrictions);
26
27         public virtual TLink Create() => Links.Create();
28
29         public virtual TLink Update(IList<TLink> restrictions) => Links.Update(restrictions);
30
31         public virtual void Delete(TLink link) => Links.Delete(link);
32
33         protected override bool AllowMultipleDisposeCalls => true;
34
35         protected override void Dispose(bool manual, bool wasDisposed)
36         {
37             if (!wasDisposed)
38             {
39                 Links.DisposeIfPossible();
40             }
41         }
42     }

```

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

```

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

```

```

25
26     public override void Delete(TLink link)
27     {
28         Links.EnsureLinkExists(link, nameof(link));
29         Links.Delete(link);
30     }
31 }
32 }

```

./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) =>
38             ↳ Links.Update(Links.ResolveConstantAsSelfReference(Constants.Itself, restrictions));
39     }
40 }

```

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

```

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

```

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

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

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

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

```

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)
12         {
13             Links.EnsureNoUsages(restrictions[Constants.IndexPart]);
14             return Links.Update(restrictions);
15         }
16
17         public override void Delete(TLink link)
18         {
19             Links.EnsureNoUsages(link);
20             Links.Delete(link);
21         }
22     }
23 }

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

./Platform.Data.Doublets/Decorators/UInt64Links.cs
1 using System;
2 using System.Collections.Generic;
3 using Platform.Collections;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Decorators
8 {
9     /// <summary>
10     /// Представляет объект для работы с базой данных (файлом) в формате Links (массива связей).
11     /// </summary>
12     /// <remarks>
13     /// Возможные оптимизации:
14     /// Объединение в одном поле Source и Target с уменьшением до 32 бит.
15     ///     + меньше объём БД
16     ///     - меньше производительность
17     ///     - больше ограничение на количество связей в БД)
18     /// Ленивое хранение размеров поддеревьев (расчитываемое по мере использования БД)
19     ///     + меньше объём БД
20     ///     - больше сложность
21     ///
22     /// Текущее теоретическое ограничение на индекс связи, из-за использования 5 бит в размере
23     ///     ↳ поддеревьев для AVL баланса и флагов нитей: 2 в степени(64 минус 5 равно 59 ) равно 576
24     ///     ↳ 460 752 303 423 488
25     /// Желательно реализовать поддержку переключения между деревьями и битовыми индексами
26     ///     ↳ (битовыми строками) - вариант матрицы (выстраиваемой лениво).
27     ///
28     /// Решить отключать ли проверки при компиляции под Release. Т.е. исключения будут
29     ///     ↳ выбрасываться только при #if DEBUG
30     /// </remarks>
31     public class UInt64Links : LinksDisposableDecoratorBase<ulong>
32     {

```

```

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() => Links.CreatePoint();
38
39 public override ulong Update(IList<ulong> restrictions)
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 (restrictions.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 = restrictions[sourcePartConstant];
59     this.EnsureLinkIsItselfOrExists(newSource,
60     ↪ $"{nameof(restrictions)}[{nameof(sourcePartConstant)}]");
61     var targetPartConstant = constants.TargetPart;
62     var newTarget = restrictions[targetPartConstant];
63     this.EnsureLinkIsItselfOrExists(newTarget,
64     ↪ $"{nameof(restrictions)}[{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(ulong linkIndex)
91 {
92     Links.EnsureLinkExists(linkIndex);
93     Links.EnforceResetValues(linkIndex);
94     this.DeleteAllUsages(linkIndex);
95     Links.Delete(linkIndex);
96 }
97 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using Platform.Collections;
5 using Platform.Collections.Arrays;
6 using Platform.Collections.Lists;
7 using Platform.Data.Universal;

```

```

8
9 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets.Decorators
12 {
13     /// <remarks>
14     /// What does empty pattern (for condition or substitution) mean? Nothing or Everything?
15     /// Now we go with nothing. And nothing is something one, but empty, and cannot be changed
16     /// ↪ by itself. But can cause creation (update from nothing) or deletion (update to nothing).
17     ///
18     /// TODO: Decide to change to IDoubletLinks or not to change. (Better to create
19     /// ↪ DefaultUniLinksBase, that contains logic itself and can be implemented using both
20     /// ↪ IDoubletLinks and ILinks.)
21     /// </remarks>
22     internal class UniLinks<TLink> : LinksDecoratorBase<TLink>, IUniLinks<TLink>
23     {
24         private static readonly EqualityComparer<TLink> _equalityComparer =
25             ↪ EqualityComparer<TLink>.Default;
26
27         public UniLinks(ILinks<TLink> links) : base(links) { }
28
29         private struct Transition
30         {
31             public IList<TLink> Before;
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<LinksCombinedConstants<TLink, TLink,
42         //    ↪ int>>.Single.Null;
43         //public static readonly IReadOnlyList<TLink> NullLink = new
44         //    ↪ ReadOnlyCollection<TLink>(new List<TLink> { NullConstant, NullConstant, NullConstant
45         //    ↪ });
46
47         // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution
48         //    ↪ (Links-Expression)
49         public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>, TLink>
50             ↪ matchedHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
51             ↪ substitutedHandler)
52         {
53             ///List<Transition> transitions = null;
54             ///if (!restriction.IsNullOrEmpty())
55             ///{
56             ///    // Есть причина делать проход (чтение)
57             ///    if (matchedHandler != null)
58             ///    {
59             ///        if (!substitution.IsNullOrEmpty())
60             ///        {
61             ///            // restriction => { 0, 0, 0 } | { 0 } // Create
62             ///            // substitution => { itself, 0, 0 } | { itself, itself, itself } //
63             ///            ↪ Create / Update
64             ///            // substitution => { 0, 0, 0 } | { 0 } // Delete
65             ///            transitions = new List<Transition>();
66             ///            if (Equals(substitution[Constants.IndexPart], Constants.Null))
67             ///            {
68             ///                // If index is Null, that means we always ignore every other
69             ///                ↪ value (they are also Null by definition)
70             ///                var matchDecision = matchedHandler(, NullLink);
71             ///                if (Equals(matchDecision, Constants.Break))
72             ///                    return false;
73             ///                if (!Equals(matchDecision, Constants.Skip))
74             ///                    transitions.Add(new Transition(matchedLink, newValue));
75             ///            }
76             ///        }
77             ///        else
78             ///        {
79             ///            Func<T, bool> handler;
80             ///            handler = link =>
81             ///            {
82             ///                var matchedLink = Memory.GetLinkValue(link);
83             ///                var newValue = Memory.GetLinkValue(link);
84             ///                newValue[Constants.IndexPart] = Constants.Itself;

```

```

72      newLink[Constants.SourcePart] =
    ↪ Equals(substitution[Constants.SourcePart], Constants.Itself) ?
    ↪ matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
73      newLink[Constants.TargetPart] =
    ↪ Equals(substitution[Constants.TargetPart], Constants.Itself) ?
    ↪ matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
74      var matchDecision = matchedHandler(matchedLink, newLink);
75      if (Equals(matchDecision, Constants.Break))
76          return false;
77      if (!Equals(matchDecision, Constants.Skip))
78          transitions.Add(new Transition(matchedLink, newLink));
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<ILink<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 (restriction.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)),
    ↪ Constants.Break), restriction))

```



```

144         //         return Constants.Break;
145     //}
146     //else // Update
147     //{
148         //     //List<IList<T>> matchedLinks = null;
149         //     if (matchedHandler != null)
150         //     {
151             //         matchedLinks = new List<IList<T>>();
152             //         Func<T, bool> handler = link =>
153             //         {
154                 //             var matchedLink = Memory.GetLinkValue(link);
155                 //             var matchDecision = matchedHandler(matchedLink);
156                 //             if (Equals(matchDecision, Constants.Break))
157                     //                 return false;
158                 //             if (!Equals(matchDecision, Constants.Skip))
159                     //                 matchedLinks.Add(matchedLink);
160                 //             return true;
161             //         };
162             //         if (!Memory.Each(handler, restriction))
163                 //             return Constants.Break;
164             //     }
165             //     if (!matchedLinks.IsNullOrEmpty())
166             //     {
167                 //         var totalMatchedLinks = matchedLinks.Count;
168                 //         for (var i = 0; i < totalMatchedLinks; i++)
169                 //         {
170                     //             var matchedLink = matchedLinks[i];
171                     //             if (substitutedHandler != null)
172                     //             {
173                         //                 var newValue = new List<T>(); // TODO: Prepare value to update here
174                         //                 // TODO: Decide is it actually needed to use Before and After
175                         //                 substitution handling.
176                         //                 var substitutedDecision = substitutedHandler(matchedLink,
177                         //                 newValue);
178                         //                 if (Equals(substitutedDecision, Constants.Break))
179                             //                     return Constants.Break;
180                         //                 if (Equals(substitutedDecision, Constants.Continue))
181                         //                 {
182                             //                     // Actual update here
183                             //                     Memory.SetLinkValue(newValue);
184                             //                 }
185                         //                 if (Equals(substitutedDecision, Constants.Skip))
186                         //                 {
187                             //                     // Cancel the update. TODO: decide use separate Cancel
188                             //                     constant or Skip is enough?
189                             //                 }
190                         //             }
191                     //         }
192             //     }
193         // }
194     // }
195     // }
196     // }
197     // }
198     // }
199     // }
200     // }
201     // }
202     // }
203     // }
204     // }
205     // }
206     // }
207     // }
208     // }
209     // }
210     // }
211     // }
212     // }
213     // }
214     // }
215     // }
216     // }
217     // }
218     // }
219     // }
220     // }
221     // }
222     // }
223     // }
224     // }
225     // }
226     // }
227     // }
228     // }
229     // }
230     // }
231     // }
232     // }
233     // }
234     // }
235     // }
236     // }
237     // }
238     // }
239     // }
240     // }
241     // }
242     // }
243     // }
244     // }
245     // }
246     // }
247     // }
248     // }
249     // }
250     // }
251     // }
252     // }
253     // }
254     // }
255     // }
256     // }
257     // }
258     // }
259     // }
260     // }
261     // }
262     // }
263     // }
264     // }
265     // }
266     // }
267     // }
268     // }
269     // }
270     // }
271     // }
272     // }
273     // }
274     // }
275     // }
276     // }
277     // }
278     // }
279     // }
280     // }
281     // }
282     // }
283     // }
284     // }
285     // }
286     // }
287     // }
288     // }
289     // }
290     // }
291     // }
292     // }
293     // }
294     // }
295     // }
296     // }
297     // }
298     // }
299     // }
300     // }
301     // }
302     // }
303     // }
304     // }
305     // }
306     // }
307     // }
308     // }
309     // }
310     // }
311     // }
312     // }
313     // }
314     // }
315     // }
316     // }
317     // }
318     // }
319     // }
320     // }
321     // }
322     // }
323     // }
324     // }
325     // }
326     // }
327     // }
328     // }
329     // }
330     // }
331     // }
332     // }
333     // }
334     // }
335     // }
336     // }
337     // }
338     // }
339     // }
340     // }
341     // }
342     // }
343     // }
344     // }
345     // }
346     // }
347     // }
348     // }
349     // }
350     // }
351     // }
352     // }
353     // }
354     // }
355     // }
356     // }
357     // }
358     // }
359     // }
360     // }
361     // }
362     // }
363     // }
364     // }
365     // }
366     // }
367     // }
368     // }
369     // }
370     // }
371     // }
372     // }
373     // }
374     // }
375     // }
376     // }
377     // }
378     // }
379     // }
380     // }
381     // }
382     // }
383     // }
384     // }
385     // }
386     // }
387     // }
388     // }
389     // }
390     // }
391     // }
392     // }
393     // }
394     // }
395     // }
396     // }
397     // }
398     // }
399     // }
400     // }
401     // }
402     // }
403     // }
404     // }
405     // }
406     // }
407     // }
408     // }
409     // }
410     // }
411     // }
412     // }
413     // }
414     // }
415     // }
416     // }
417     // }
418     // }
419     // }
420     // }
421     // }
422     // }
423     // }
424     // }
425     // }
426     // }
427     // }
428     // }
429     // }
430     // }
431     // }
432     // }
433     // }
434     // }
435     // }
436     // }
437     // }
438     // }
439     // }
440     // }
441     // }
442     // }
443     // }
444     // }
445     // }
446     // }
447     // }
448     // }
449     // }
450     // }
451     // }
452     // }
453     // }
454     // }
455     // }
456     // }
457     // }
458     // }
459     // }
460     // }
461     // }
462     // }
463     // }
464     // }
465     // }
466     // }
467     // }
468     // }
469     // }
470     // }
471     // }
472     // }
473     // }
474     // }
475     // }
476     // }
477     // }
478     // }
479     // }
480     // }
481     // }
482     // }
483     // }
484     // }
485     // }
486     // }
487     // }
488     // }
489     // }
490     // }
491     // }
492     // }
493     // }
494     // }
495     // }
496     // }
497     // }
498     // }
499     // }
500     // }
501     // }
502     // }
503     // }
504     // }
505     // }
506     // }
507     // }
508     // }
509     // }
510     // }
511     // }
512     // }
513     // }
514     // }
515     // }
516     // }
517     // }
518     // }
519     // }
520     // }
521     // }
522     // }
523     // }
524     // }
525     // }
526     // }
527     // }
528     // }
529     // }
530     // }
531     // }
532     // }
533     // }
534     // }
535     // }
536     // }
537     // }
538     // }
539     // }
540     // }
541     // }
542     // }
543     // }
544     // }
545     // }
546     // }
547     // }
548     // }
549     // }
550     // }
551     // }
552     // }
553     // }
554     // }
555     // }
556     // }
557     // }
558     // }
559     // }
560     // }
561     // }
562     // }
563     // }
564     // }
565     // }
566     // }
567     // }
568     // }
569     // }
570     // }
571     // }
572     // }
573     // }
574     // }
575     // }
576     // }
577     // }
578     // }
579     // }
580     // }
581     // }
582     // }
583     // }
584     // }
585     // }
586     // }
587     // }
588     // }
589     // }
590     // }
591     // }
592     // }
593     // }
594     // }
595     // }
596     // }
597     // }
598     // }
599     // }
600     // }
601     // }
602     // }
603     // }
604     // }
605     // }
606     // }
607     // }
608     // }
609     // }
610     // }
611     // }
612     // }
613     // }
614     // }
615     // }
616     // }
617     // }
618     // }
619     // }
620     // }
621     // }
622     // }
623     // }
624     // }
625     // }
626     // }
627     // }
628     // }
629     // }
630     // }
631     // }
632     // }
633     // }
634     // }
635     // }
636     // }
637     // }
638     // }
639     // }
640     // }
641     // }
642     // }
643     // }
644     // }
645     // }
646     // }
647     // }
648     // }
649     // }
650     // }
651     // }
652     // }
653     // }
654     // }
655     // }
656     // }
657     // }
658     // }
659     // }
660     // }
661     // }
662     // }
663     // }
664     // }
665     // }
666     // }
667     // }
668     // }
669     // }
670     // }
671     // }
672     // }
673     // }
674     // }
675     // }
676     // }
677     // }
678     // }
679     // }
680     // }
681     // }
682     // }
683     // }
684     // }
685     // }
686     // }
687     // }
688     // }
689     // }
690     // }
691     // }
692     // }
693     // }
694     // }
695     // }
696     // }
697     // }
698     // }
699     // }
700     // }
701     // }
702     // }
703     // }
704     // }
705     // }
706     // }
707     // }
708     // }
709     // }
710     // }
711     // }
712     // }
713     // }
714     // }
715     // }
716     // }
717     // }
718     // }
719     // }
720     // }
721     // }
722     // }
723     // }
724     // }
725     // }
726     // }
727     // }
728     // }
729     // }
730     // }
731     // }
732     // }
733     // }
734     // }
735     // }
736     // }
737     // }
738     // }
739     // }
740     // }
741     // }
742     // }
743     // }
744     // }
745     // }
746     // }
747     // }
748     // }
749     // }
750     // }
751     // }
752     // }
753     // }
754     // }
755     // }
756     // }
757     // }
758     // }
759     // }
760     // }
761     // }
762     // }
763     // }
764     // }
765     // }
766     // }
767     // }
768     // }
769     // }
770     // }
771     // }
772     // }
773     // }
774     // }
775     // }
776     // }
777     // }
778     // }
779     // }
780     // }
781     // }
782     // }
783     // }
784     // }
785     // }
786     // }
787     // }
788     // }
789     // }
790     // }
791     // }
792     // }
793     // }
794     // }
795     // }
796     // }
797     // }
798     // }
799     // }
800     // }
801     // }
802     // }
803     // }
804     // }
805     // }
806     // }
807     // }
808     // }
809     // }
810     // }
811     // }
812     // }
813     // }
814     // }
815     // }
816     // }
817     // }
818     // }
819     // }
820     // }
821     // }
822     // }
823     // }
824     // }
825     // }
826     // }
827     // }
828     // }
829     // }
830     // }
831     // }
832     // }
833     // }
834     // }
835     // }
836     // }
837     // }
838     // }
839     // }
840     // }
841     // }
842     // }
843     // }
844     // }
845     // }
846     // }
847     // }
848     // }
849     // }
850     // }
851     // }
852     // }
853     // }
854     // }
855     // }
856     // }
857     // }
858     // }
859     // }
860     // }
861     // }
862     // }
863     // }
864     // }
865     // }
866     // }
867     // }
868     // }
869     // }
870     // }
871     // }
872     // }
873     // }
874     // }
875     // }
876     // }
877     // }
878     // }
879     // }
880     // }
881     // }
882     // }
883     // }
884     // }
885     // }
886     // }
887     // }
888     // }
889     // }
890     // }
891     // }
892     // }
893     // }
894     // }
895     // }
896     // }
897     // }
898     // }
899     // }
900     // }
901     // }
902     // }
903     // }
904     // }
905     // }
906     // }
907     // }
908     // }
909     // }
910     // }
911     // }
912     // }
913     // }
914     // }
915     // }
916     // }
917     // }
918     // }
919     // }
920     // }
921     // }
922     // }
923     // }
924     // }
925     // }
926     // }
927     // }
928     // }
929     // }
930     // }
931     // }
932     // }
933     // }
934     // }
935     // }
936     // }
937     // }
938     // }
939     // }
940     // }
941     // }
942     // }
943     // }
944     // }
945     // }
946     // }
947     // }
948     // }
949     // }
950     // }
951     // }
952     // }
953     // }
954     // }
955     // }
956     // }
957     // }
958     // }
959     // }
960     // }
961     // }
962     // }
963     // }
964     // }
965     // }
966     // }
967     // }
968     // }
969     // }
970     // }
971     // }
972     // }
973     // }
974     // }
975     // }
976     // }
977     // }
978     // }
979     // }
980     // }
981     // }
982     // }
983     // }
984     // }
985     // }
986     // }
987     // }
988     // }
989     // }
990     // }
991     // }
992     // }
993     // }
994     // }
995     // }
996     // }
997     // }
998     // }
999     // }
1000    // }

```

```

214     if (_equalityComparer.Equals(after[0], default))
215     {
216         var newLink = Links.Create();
217         after[0] = newLink;
218     }
219     if (substitution.Count == 1)
220     {
221         after = Links.GetLink(substitution[0]);
222     }
223     else if (substitution.Count == 3)
224     {
225         Links.Update(after);
226     }
227     else
228     {
229         throw new NotSupportedException();
230     }
231     if (matchHandler != null)
232     {
233         return substitutionHandler(before, after);
234     }
235     return Constants.Continue;
236 }
237 else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
238 {
239     if (patternOrCondition.Count == 1)
240     {
241         var linkToDelete = patternOrCondition[0];
242         var before = Links.GetLink(linkToDelete);
243         if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
244             ↪ Constants.Break))
245         {
246             return Constants.Break;
247         }
248         var after = ArrayPool<TLink>.Empty;
249         Links.Update(linkToDelete, Constants.Null, Constants.Null);
250         Links.Delete(linkToDelete);
251         if (matchHandler != null)
252         {
253             return substitutionHandler(before, after);
254         }
255         return Constants.Continue;
256     }
257     else
258     {
259         throw new NotSupportedException();
260     }
261 }
262 else // Replace / Update
263 {
264     if (patternOrCondition.Count == 1) //-V3125
265     {
266         var linkToUpdate = patternOrCondition[0];
267         var before = Links.GetLink(linkToUpdate);
268         if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
269             ↪ Constants.Break))
270         {
271             return Constants.Break;
272         }
273         var after = (IList<TLink>)substitution.ToArray(); //-V3125
274         if (_equalityComparer.Equals(after[0], default))
275         {
276             after[0] = linkToUpdate;
277         }
278         if (substitution.Count == 1)
279         {
280             if (!_equalityComparer.Equals(substitution[0], linkToUpdate))
281             {
282                 after = Links.GetLink(substitution[0]);
283                 Links.Update(linkToUpdate, Constants.Null, Constants.Null);
284                 Links.Delete(linkToUpdate);
285             }
286         }
287         else if (substitution.Count == 3)
288         {
289             Links.Update(after);
290         }
291         else

```



```

11
12     public T Source { get; set; }
13     public T Target { get; set; }
14
15     public Doublet(T source, T target)
16     {
17         Source = source;
18         Target = target;
19     }
20
21     public override string ToString() => $"{Source}->{Target}";
22
23     public bool Equals(Doublet<T> other) => _equalityComparer.Equals(Source, other.Source)
24         ↳ && _equalityComparer.Equals(Target, other.Target);
25
26     public override bool Equals(object obj) => obj is Doublet<T> doublet ?
27         ↳ base.Equals(doublet) : false;
28
29     public override int GetHashCode() => (Source, Target).GetHashCode();
30 }

```

./Platform.Data.Doublets/Hybrid.cs

```

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

```

```

49         var unsignedMethod = typeof(To).GetTypeInfo().GetMethod("Unsigned", new[] {
50             ↳ signedVersion });
51         emitter.Call(unsignedMethod);
52         emitter.Return();
53     });
54 }
55 public readonly T Value;
56 public bool IsNothing => Convert.ToInt64(To.Signed(Value)) == 0;
57 public bool IsInternal => Convert.ToInt64(To.Signed(Value)) > 0;
58 public bool IsExternal => Convert.ToInt64(To.Signed(Value)) < 0;
59 public long AbsoluteValue =>
60     ↳ Platform.Numbers.Math.Abs(Convert.ToInt64(To.Signed(Value)));
61 public Hybrid(T value)
62 {
63     Ensure.OnDebug.IsUnsignedInteger<T>();
64     Value = value;
65 }
66
67 public Hybrid(object value) => Value = To.UnsignedAs<T>(Convert.ChangeType(value,
68     ↳ Type<T>.SignedVersion));
69
70 public Hybrid(object value, bool isExternal)
71 {
72     //var signedType = Type<T>.SignedVersion;
73     //var signedValue = Convert.ChangeType(value, signedType);
74     //var abs = typeof(Platform.Numbers.Math).GetTypeInfo().GetMethod("Abs").MakeGeneric
75     ↳ Method(signedType);
76     //var negate = typeof(Platform.Numbers.Math).GetTypeInfo().GetMethod("Negate").MakeG
77     ↳ enericMethod(signedType);
78     //var absoluteValue = abs.Invoke(null, new[] { signedValue });
79     //var resultValue = isExternal ? negate.Invoke(null, new[] { absoluteValue }) :
80     ↳ absoluteValue;
81     //Value = To.UnsignedAs<T>(resultValue);
82     if (isExternal)
83     {
84         Value = _absAndNegateAndConvert(value);
85     }
86     else
87     {
88         Value = _absAndConvert(value);
89     }
90 }
91
92 public static implicit operator Hybrid<T>(T integer) => new Hybrid<T>(integer);
93
94 public static explicit operator Hybrid<T>(ulong integer) => new Hybrid<T>(integer);
95
96 public static explicit operator Hybrid<T>(long integer) => new Hybrid<T>(integer);
97
98 public static explicit operator Hybrid<T>(uint integer) => new Hybrid<T>(integer);
99
100 public static explicit operator Hybrid<T>(int integer) => new Hybrid<T>(integer);
101
102 public static explicit operator Hybrid<T>(ushort integer) => new Hybrid<T>(integer);
103
104 public static explicit operator Hybrid<T>(short integer) => new Hybrid<T>(integer);
105
106 public static explicit operator Hybrid<T>(byte integer) => new Hybrid<T>(integer);
107
108 public static explicit operator Hybrid<T>(sbyte integer) => new Hybrid<T>(integer);
109
110 public static implicit operator T(Hybrid<T> hybrid) => hybrid.Value;
111
112 public static explicit operator ulong(Hybrid<T> hybrid) =>
113     ↳ Convert.ToUInt64(hybrid.Value);
114
115 public static explicit operator long(Hybrid<T> hybrid) => hybrid.AbsoluteValue;
116
117 public static explicit operator uint(Hybrid<T> hybrid) => Convert.ToUInt32(hybrid.Value);
118
119 public static explicit operator int(Hybrid<T> hybrid) =>
120     ↳ Convert.ToInt32(hybrid.AbsoluteValue);
121
122 public static explicit operator ushort(Hybrid<T> hybrid) =>
123     ↳ Convert.ToUInt16(hybrid.Value);

```

```

118     public static explicit operator short(Hybrid<T> hybrid) =>
119         ↪ Convert.ToInt16(hybrid.AbsoluteValue);
120
121     public static explicit operator byte(Hybrid<T> hybrid) => Convert.ToByte(hybrid.Value);
122
123     public static explicit operator sbyte(Hybrid<T> hybrid) =>
124         ↪ Convert.ToSByte(hybrid.AbsoluteValue);
125
126     public override string ToString() => IsNothing ? default(T) == null ? "Nothing" :
127         ↪ default(T).ToString() : IsExternal ? $"{<AbsoluteValue>}" : Value.ToString();
128 }
129 }

```

./Platform.Data.Doublets/ILinks.cs

```

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

```

./Platform.Data.Doublets/ILinksExtensions.cs

```

1 using System;
2 using System.Collections;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Runtime.CompilerServices;
6 using Platform.Ranges;
7 using Platform.Collections.Arrays;
8 using Platform.Numbers;
9 using Platform.Random;
10 using Platform.Setters;
11 using Platform.Data.Exceptions;
12 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());
37                     Integer<TLink> source = RandomHelpers.Default.NextUInt64(linkAddressRange);
38                     Integer<TLink> target = RandomHelpers.Default.NextUInt64(linkAddressRange);
39                     links.SearchOrDefault(source, target);
40                 }
41
42             public static void RunRandomDeletions<TLink>(this ILinks<TLink> links, long
43                 ↪ amountOfDeletions)
44             {
45                 var min = (ulong)amountOfDeletions > (Integer<TLink>)links.Count() ? 1 :
46                     ↪ (Integer<TLink>)links.Count() - (ulong)amountOfDeletions;
47                 for (long i = 0; i < amountOfDeletions; i++)
48                 {
49                     var linksAddressRange = new Range<ulong>(min, (Integer<TLink>)links.Count());
50                     Integer<TLink> link = RandomHelpers.Default.NextUInt64(linksAddressRange);
51                     links.Delete(link);
52                 }
53             }
54         }
55     }
56 }

```

```

50         if ((Integer<TLink>)links.Count() < min)
51         {
52             break;
53         }
54     }
55 }
56
57 /// <remarks>
58 /// TODO: Возможно есть очень простой способ это сделать.
59 /// (Например просто удалить файл, или изменить его размер таким образом,
60 /// чтобы удалился весь контент)
61 /// Например через _header->AllocatedLinks в ResizableDirectMemoryLinks
62 /// </remarks>
63 public static void DeleteAll<TLink>(this ILinks<TLink> links)
64 {
65     var equalityComparer = EqualityComparer<TLink>.Default;
66     var comparer = Comparer<TLink>.Default;
67     for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
        ↪ Arithmetic.Decrement(i))
68     {
69         links.Delete(i);
70         if (!equalityComparer.Equals(links.Count(), Arithmetic.Decrement(i)))
71         {
72             i = links.Count();
73         }
74     }
75 }
76
77 public static TLink First<TLink>(this ILinks<TLink> links)
78 {
79     TLink firstLink = default;
80     var equalityComparer = EqualityComparer<TLink>.Default;
81     if (equalityComparer.Equals(links.Count(), default))
82     {
83         throw new Exception("В хранилище нет связей.");
84     }
85     links.Each(links.Constants.Any, links.Constants.Any, link =>
86     {
87         firstLink = link[links.Constants.IndexPart];
88         return links.Constants.Break;
89     });
90     if (equalityComparer.Equals(firstLink, default))
91     {
92         throw new Exception("В процессе поиска по хранилищу не было найдено связей.");
93     }
94     return firstLink;
95 }
96
97 public static bool IsInnerReference<TLink>(this ILinks<TLink> links, TLink reference)
98 {
99     var constants = links.Constants;
100     var comparer = Comparer<TLink>.Default;
101     return comparer.Compare(constants.MinPossibleIndex, reference) >= 0 &&
        ↪ comparer.Compare(reference, constants.MaxPossibleIndex) <= 0;
102 }
103
104 #region Paths
105
106 /// <remarks>
107 /// TODO: Как так? Как то что ниже может быть корректно?
108 /// Скорее всего практически не применимо
109 /// Предполагалось, что можно было конвертировать формируемый в проходе через
        ↪ SequenceWalker
110 /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
111 /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
112 /// </remarks>
113 public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[]
        ↪ path)
114 {
115     var current = path[0];
116     //EnsureLinkExists(current, "path");
117     if (!links.Exists(current))
118     {
119         return false;
120     }
121     var equalityComparer = EqualityComparer<TLink>.Default;
122     var constants = links.Constants;
123     for (var i = 1; i < path.Length; i++)
124     {

```

```

125     var next = path[i];
126     var values = links.GetLink(current);
127     var source = values[constants.SourcePart];
128     var target = values[constants.TargetPart];
129     if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
130         ↪ next))
131     {
132         //throw new Exception(string.Format("Невозможно выбрать путь, так как и
133         ↪ Source и Target совпадают с элементом пути {0}.", next));
134         return false;
135     }
136     if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
137         ↪ target))
138     {
139         //throw new Exception(string.Format("Невозможно продолжить путь через
140         ↪ элемент пути {0}", next));
141         return false;
142     }
143     current = next;
144 }
145 return true;
146 }
147
148 /// <remarks>
149 /// Может потребовать дополнительного стека для PathElement's при использовании
150 ↪ SequenceWalker.
151 /// </remarks>
152 public static TLink GetByKeyes<TLink>(this ILinks<TLink> links, TLink root, params int[]
153     ↪ path)
154 {
155     links.EnsureLinkExists(root, "root");
156     var currentLink = root;
157     for (var i = 0; i < path.Length; i++)
158     {
159         currentLink = links.GetLink(currentLink)[path[i]];
160     }
161     return currentLink;
162 }
163
164 public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this ILinks<TLink>
165     ↪ links, TLink root, ulong size, ulong index)
166 {
167     var constants = links.Constants;
168     var source = constants.SourcePart;
169     var target = constants.TargetPart;
170     if (!Platform.Numbers.Math.IsPowerOfTwo(size))
171     {
172         throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes other
173         ↪ than powers of two are not supported.");
174     }
175     var path = new BitArray(BitConverter.GetBytes(index));
176     var length = Bit.GetLowestPosition(size);
177     links.EnsureLinkExists(root, "root");
178     var currentLink = root;
179     for (var i = length - 1; i >= 0; i--)
180     {
181         currentLink = links.GetLink(currentLink)[path[i] ? target : source];
182     }
183     return currentLink;
184 }
185
186 #endregion
187
188 /// <summary>
189 /// Возвращает индекс указанной связи.
190 /// </summary>
191 /// <param name="links">Хранилище связей.</param>
192 /// <param name="link">Связь представленная списком, состоящим из её адреса и
193     ↪ содержимого.</param>
194 /// <returns>Индекс начальной связи для указанной связи.</returns>
195 [MethodImpl(MethodImplOptions.AggressiveInlining)]
196 public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
197     ↪ link[links.Constants.IndexPart];
198
199 /// <summary>
200 /// Возвращает индекс начальной (Source) связи для указанной связи.
201 /// </summary>
202 /// <param name="links">Хранилище связей.</param>

```



```

193 /// <param name="link">Индекс связи.</param>
194 /// <returns>Индекс начальной связи для указанной связи.</returns>
195 [MethodImpl(MethodImplOptions.AggressiveInlining)]
196 public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink link) =>
197     ↪ links.GetLink(link)[links.Constants.SourcePart];
198
199 /// <summary>
200 /// Возвращает индекс начальной (Source) связи для указанной связи.
201 /// </summary>
202 /// <param name="links">Хранилище связей.</param>
203 /// <param name="link">Связь представленная списком, состоящим из её адреса и
204     ↪ содержимого.</param>
205 /// <returns>Индекс начальной связи для указанной связи.</returns>
206 [MethodImpl(MethodImplOptions.AggressiveInlining)]
207 public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
208     ↪ link[links.Constants.SourcePart];
209
210 /// <summary>
211 /// Возвращает индекс конечной (Target) связи для указанной связи.
212 /// </summary>
213 /// <param name="links">Хранилище связей.</param>
214 /// <param name="link">Индекс связи.</param>
215 /// <returns>Индекс конечной связи для указанной связи.</returns>
216 [MethodImpl(MethodImplOptions.AggressiveInlining)]
217 public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
218     ↪ links.GetLink(link)[links.Constants.TargetPart];
219
220 /// <summary>
221 /// Возвращает индекс конечной (Target) связи для указанной связи.
222 /// </summary>
223 /// <param name="links">Хранилище связей.</param>
224 /// <param name="link">Связь представленная списком, состоящим из её адреса и
225     ↪ содержимого.</param>
226 /// <returns>Индекс конечной связи для указанной связи.</returns>
227 [MethodImpl(MethodImplOptions.AggressiveInlining)]
228 public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
229     ↪ link[links.Constants.TargetPart];
230
231 /// <summary>
232 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
233     ↪ (handler) для каждой подходящей связи.
234 /// </summary>
235 /// <param name="links">Хранилище связей.</param>
236 /// <param name="handler">Обработчик каждой подходящей связи.</param>
237 /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
238     ↪ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
239     ↪ Any - отсутствие ограничения, 1..∞ конкретный адрес 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, Func<IList<TLink>, TLink>
244     ↪ handler, params TLink[] restrictions)
245     => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
246         ↪ links.Constants.Continue);
247
248 /// <summary>
249 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
250     ↪ (handler) для каждой подходящей связи.
251 /// </summary>
252 /// <param name="links">Хранилище связей.</param>
253 /// <param name="source">Значение, определяющее соответствующие шаблону связи.
254     ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
255     ↪ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
256 /// <param name="target">Значение, определяющее соответствующие шаблону связи.
257     ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
258     ↪ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
259 /// <param name="handler">Обработчик каждой подходящей связи.</param>
260 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
261     ↪ случае.</returns>
262 [MethodImpl(MethodImplOptions.AggressiveInlining)]
263 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
264     ↪ Func<TLink, bool> handler)
265 {
266     var constants = links.Constants;
267     return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
268         ↪ 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)]
311     public static bool Exists<TLink>(this ILinks<TLink> links, TLink source, TLink target)
312     {
313         => Comparer<TLink>.Default.Compare(links.Count(links.Constants.Any, source, target),
314             default) > 0;
315     }
316
317     #region Ensure
318     // TODO: May be move to EnsureExtensions or make it both there and here
319
320     [MethodImpl(MethodImplOptions.AggressiveInlining)]
321     public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
322     reference, string argumentName)
323     {
324         if (links.IsInnerReference(reference) && !links.Exists(reference))
325         {
326             throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
327         }
328     }

```

```

312     }
313
314     [MethodImpl(MethodImplOptions.AggressiveInlining)]
315     public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
316     ↪ IList<TLink> restrictions, string argumentName)
317     {
318         for (int i = 0; i < restrictions.Count; i++)
319         {
320             links.EnsureInnerReferenceExists(restrictions[i], argumentName);
321         }
322     }
323
324     [MethodImpl(MethodImplOptions.AggressiveInlining)]
325     public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
326     ↪ restrictions)
327     {
328         for (int i = 0; i < restrictions.Count; i++)
329         {
330             links.EnsureLinkIsAnyOrExists(restrictions[i], nameof(restrictions));
331         }
332     }
333
334     [MethodImpl(MethodImplOptions.AggressiveInlining)]
335     public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
336     ↪ string argumentName)
337     {
338         var equalityComparer = EqualityComparer<TLink>.Default;
339         if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
340         {
341             throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
342         }
343     }
344
345     [MethodImpl(MethodImplOptions.AggressiveInlining)]
346     public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links, TLink
347     ↪ link, string argumentName)
348     {
349         var equalityComparer = EqualityComparer<TLink>.Default;
350         if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
351         {
352             throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
353         }
354     }
355
356     /// <param name="links">Хранилище связей.</param>
357     [MethodImpl(MethodImplOptions.AggressiveInlining)]
358     public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink source,
359     ↪ TLink target)
360     {
361         if (links.Exists(source, target))
362         {
363             throw new LinkWithSameValueAlreadyExistsException();
364         }
365     }
366
367     /// <param name="links">Хранилище связей.</param>
368     public static void EnsureNoUsages<TLink>(this ILinks<TLink> links, TLink link)
369     {
370         if (links.HasUsages(link))
371         {
372             throw new ArgumentLinkHasDependenciesException<TLink>(link);
373         }
374     }
375
376     /// <param name="links">Хранилище связей.</param>
377     public static void EnsureCreated<TLink>(this ILinks<TLink> links, params TLink[]
378     ↪ addresses) => links.EnsureCreated(links.Create, addresses);
379
380     /// <param name="links">Хранилище связей.</param>
381     public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params TLink[]
382     ↪ addresses) => links.EnsureCreated(links.CreatePoint, addresses);
383
384     /// <param name="links">Хранилище связей.</param>
385     public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
386     ↪ params TLink[] addresses)
387     {
388         var constants = links.Constants;

```

```

381     var nonExistentAddresses = new HashSet<ulong>(addresses.Where(x =>
382         ↪ !links.Exists(x)).Select(x => (ulong)(Integer<TLink>)x));
383     if (nonExistentAddresses.Count > 0)
384     {
385         var max = nonExistentAddresses.Max();
386         // TODO: Эту верхнюю границу нужно разрешить переопределять (проверить
387         ↪ применяется ли эта логика)
388         max = System.Math.Min(max, (Integer<TLink>)constants.MaxPossibleIndex);
389         var createdLinks = new List<TLink>();
390         var equalityComparer = EqualityComparer<TLink>.Default;
391         TLink createdLink = creator();
392         while (!equalityComparer.Equals(createdLink, (Integer<TLink>)max))
393         {
394             createdLinks.Add(createdLink);
395         }
396         for (var i = 0; i < createdLinks.Count; i++)
397         {
398             if (!nonExistentAddresses.Contains((Integer<TLink>)createdLinks[i]))
399             {
400                 links.Delete(createdLinks[i]);
401             }
402         }
403     }
404     #endregion
405
406     /// <param name="links">Хранилище связей.</param>
407     public static ulong CountUsages<TLink>(this ILinks<TLink> links, TLink link)
408     {
409         var constants = links.Constants;
410         var values = links.GetLink(link);
411         ulong usagesAsSource = (Integer<TLink>)links.Count(new Link<TLink>(constants.Any,
412             ↪ link, constants.Any));
413         var equalityComparer = EqualityComparer<TLink>.Default;
414         if (equalityComparer.Equals(values[constants.SourcePart], link))
415         {
416             usagesAsSource--;
417         }
418         ulong usagesAsTarget = (Integer<TLink>)links.Count(new Link<TLink>(constants.Any,
419             ↪ constants.Any, link));
420         if (equalityComparer.Equals(values[constants.TargetPart], link))
421         {
422             usagesAsTarget--;
423         }
424         return usagesAsSource + usagesAsTarget;
425     }
426
427     /// <param name="links">Хранилище связей.</param>
428     [MethodImpl(MethodImplOptions.AggressiveInlining)]
429     public static bool HasUsages<TLink>(this ILinks<TLink> links, TLink link) =>
430     ↪ links.CountUsages(link) > 0;
431
432     /// <param name="links">Хранилище связей.</param>
433     [MethodImpl(MethodImplOptions.AggressiveInlining)]
434     public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source,
435     ↪ TLink target)
436     {
437         var constants = links.Constants;
438         var values = links.GetLink(link);
439         var equalityComparer = EqualityComparer<TLink>.Default;
440         return equalityComparer.Equals(values[constants.SourcePart], source) &&
441             ↪ equalityComparer.Equals(values[constants.TargetPart], target);
442     }
443
444     /// <summary>
445     /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
446     /// </summary>
447     /// <param name="links">Хранилище связей.</param>
448     /// <param name="source">Индекс связи, которая является началом для искомой
449     ↪ связи.</param>
450     /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
451     /// <returns>Индекс искомой связи с указанными Source (началом) и Target
452     ↪ (концом).</returns>
453     [MethodImpl(MethodImplOptions.AggressiveInlining)]
454     public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
455     ↪ target)
456     {

```

```

449     var constants = links.Constants;
450     var setter = new Setter<TLink, TLink>(constants.Continue, constants.Break, default);
451     links.Each(setter.SetFirstAndReturnFalse, constants.Any, source, target);
452     return setter.Result;
453 }
454
455 /// <param name="links">Хранилище связей.</param>
456 [MethodImpl(MethodImplOptions.AggressiveInlining)]
457 public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
458 {
459     var link = links.Create();
460     return links.Update(link, link, link);
461 }
462
463 /// <param name="links">Хранилище связей.</param>
464 [MethodImpl(MethodImplOptions.AggressiveInlining)]
465 public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
    ↪ target) => links.Update(links.Create(), source, target);
466
467 /// <summary>
468 /// Обновляет связь с указанными началом (Source) и концом (Target)
469 /// на связь с указанными началом (NewSource) и концом (NewTarget).
470 /// </summary>
471 /// <param name="links">Хранилище связей.</param>
472 /// <param name="link">Индекс обновляемой связи.</param>
473 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
    ↪ выполняется обновление.</param>
474 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
    ↪ выполняется обновление.</param>
475 /// <returns>Индекс обновлённой связи.</returns>
476 [MethodImpl(MethodImplOptions.AggressiveInlining)]
477 public static TLink Update<TLink>(this ILinks<TLink> links, TLink link, TLink newSource,
    ↪ TLink newTarget) => links.Update(new Link<TLink>(link, newSource, newTarget));
478
479 /// <summary>
480 /// Обновляет связь с указанными началом (Source) и концом (Target)
481 /// на связь с указанными началом (NewSource) и концом (NewTarget).
482 /// </summary>
483 /// <param name="links">Хранилище связей.</param>
484 /// <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>
485 /// <returns>Индекс обновлённой связи.</returns>
486 [MethodImpl(MethodImplOptions.AggressiveInlining)]
487 public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
488 {
489     if (restrictions.Length == 2)
490     {
491         return links.MergeAndDelete(restrictions[0], restrictions[1]);
492     }
493     if (restrictions.Length == 4)
494     {
495         return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
            ↪ restrictions[2], restrictions[3]);
496     }
497     else
498     {
499         return links.Update(restrictions);
500     }
501 }
502
503 [MethodImpl(MethodImplOptions.AggressiveInlining)]
504 public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
    ↪ links, TLink constant, IList<TLink> restrictions)
505 {
506     var equalityComparer = EqualityComparer<TLink>.Default;
507     var constants = links.Constants;
508     var index = restrictions[constants.IndexPart];
509     var source = restrictions[constants.SourcePart];
510     var target = restrictions[constants.TargetPart];
511     source = equalityComparer.Equals(source, constant) ? index : source;
512     target = equalityComparer.Equals(target, constant) ? index : target;
513     return new Link<TLink>(index, source, target);
514 }
515
516 /// <summary>

```

```

517     /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
518     ↪ с указанными Source (началом) и Target (концом).
519     /// </summary>
520     /// <param name="links">Хранилище связей.</param>
521     /// <param name="source">Индекс связи, которая является началом на создаваемой
522     ↪ связи.</param>
523     /// <param name="target">Индекс связи, которая является концом для создаваемой
524     ↪ связи.</param>
525     /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
526     [MethodImpl(MethodImplOptions.AggressiveInlining)]
527     public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
528     ↪ 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     /// <summary>
538     /// Обновляет связь с указанными началом (Source) и концом (Target)
539     /// 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
543     ↪ связи.</param>
544     /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
545     /// <param name="newSource">Индекс связи, которая является началом связи, на которую
546     ↪ выполняется обновление.</param>
547     /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
548     ↪ выполняется обновление.</param>
549     /// <returns>Индекс обновлённой связи.</returns>
550     [MethodImpl(MethodImplOptions.AggressiveInlining)]
551     public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
552     ↪ TLink target, TLink newSource, TLink newTarget)
553     {
554         var equalityComparer = EqualityComparer<TLink>.Default;
555         var link = links.SearchOrDefault(source, target);
556         if (equalityComparer.Equals(link, default))
557         {
558             return links.CreateAndUpdate(newSource, newTarget);
559         }
560         if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
561     ↪ target))
562         {
563             return link;
564         }
565         return links.Update(link, newSource, newTarget);
566     }
567     /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
568     /// <param name="links">Хранилище связей.</param>
569     /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
570     /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
571     [MethodImpl(MethodImplOptions.AggressiveInlining)]
572     public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
573     ↪ target)
574     {
575         var link = links.SearchOrDefault(source, target);
576         if (!EqualityComparer<TLink>.Default.Equals(link, default))
577         {
578             links.Delete(link);
579             return link;
580         }
581         return default;
582     }
583     /// <summary>Удаляет несколько связей.</summary>
584     /// <param name="links">Хранилище связей.</param>
585     /// <param name="deletedLinks">Список адресов связей к удалению.</param>
586     [MethodImpl(MethodImplOptions.AggressiveInlining)]
587     public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
588     {
589         for (int i = 0; i < deletedLinks.Count; i++)
590         {

```

```

584         links.Delete(deletedLinks[i]);
585     }
586 }
587
588 /// <remarks>Before execution of this method ensure that deleted link is detached (all
589 ↪ values - source and target are reset to null) or it might enter into infinite
590 ↪ recursion.</remarks>
591 public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
592 {
593     var anyConstant = links.Constants.Any;
594     var usagesAsSourceQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
595     links.DeleteByQuery(usagesAsSourceQuery);
596     var usagesAsTargetQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
597     links.DeleteByQuery(usagesAsTargetQuery);
598 }
599
600 public static void DeleteByQuery<TLink>(this ILinks<TLink> links, Link<TLink> query)
601 {
602     var count = (Integer<TLink>)links.Count(query);
603     if (count > 0)
604     {
605         var queryResult = new TLink[count];
606         var queryResultFiller = new ArrayFiller<TLink, TLink>(queryResult,
607             ↪ links.Constants.Continue);
608         links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
609         for (var i = (long)count - 1; i >= 0; i--)
610         {
611             links.Delete(queryResult[i]);
612         }
613     }
614 }
615
616 // TODO: Move to Platform.Data
617 public static bool AreValuesReset<TLink>(this ILinks<TLink> links, TLink linkIndex)
618 {
619     var nullConstant = links.Constants.Null;
620     var equalityComparer = EqualityComparer<TLink>.Default;
621     var link = links.GetLink(linkIndex);
622     for (int i = 1; i < link.Count; i++)
623     {
624         if (!equalityComparer.Equals(link[i], nullConstant))
625         {
626             return false;
627         }
628     }
629     return true;
630 }
631
632 // TODO: Create a universal version of this method in Platform.Data (with using of for
633 ↪ 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;

```

```

655     var usagesAsSourceQuery = new Link<TLink>(constants.Any, oldLinkIndex,
656         ↪ constants.Any);
657     long usagesAsSourceCount = (Integer<TLink>)links.Count(usagesAsSourceQuery);
658     var usagesAsTargetQuery = new Link<TLink>(constants.Any, constants.Any,
659         ↪ oldLinkIndex);
660     long usagesAsTargetCount = (Integer<TLink>)links.Count(usagesAsTargetQuery);
661     var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(oldLinkIndex)) &&
662         ↪ usagesAsSourceCount == 1 && usagesAsTargetCount == 1;
663     if (!isStandalonePoint)
664     {
665         var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
666         if (totalUsages > 0)
667         {
668             var usages = ArrayPool.Allocate<TLink>(totalUsages);
669             var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
670                 ↪ links.Constants.Continue);
671             var i = 0L;
672             if (usagesAsSourceCount > 0)
673             {
674                 links.Each(usagesFiller.AddFirstAndReturnConstant,
675                     ↪ usagesAsSourceQuery);
676                 for (; i < usagesAsSourceCount; i++)
677                 {
678                     var usage = usages[i];
679                     if (!equalityComparer.Equals(usage, oldLinkIndex))
680                     {
681                         links.Update(usage, newLinkIndex, links.GetTarget(usage));
682                     }
683                 }
684             }
685             if (usagesAsTargetCount > 0)
686             {
687                 links.Each(usagesFiller.AddFirstAndReturnConstant,
688                     ↪ usagesAsTargetQuery);
689                 for (; i < usages.Length; i++)
690                 {
691                     var usage = usages[i];
692                     if (!equalityComparer.Equals(usage, oldLinkIndex))
693                     {
694                         links.Update(usage, links.GetSource(usage), newLinkIndex);
695                     }
696                 }
697             }
698             ArrayPool.Free(usages);
699         }
700     }
701     return newLinkIndex;
702 }
703
704 /// <summary>
705 /// Replace one link with another (replaced link is deleted, children are updated or
706 /// ↪ deleted).
707 /// </summary>
708 [MethodImpl(MethodImplOptions.AggressiveInlining)]
709 public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
710     ↪ TLink newLinkIndex)
711 {
712     var equalityComparer = EqualityComparer<TLink>.Default;
713     if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
714     {
715         links.MergeUsages(oldLinkIndex, newLinkIndex);
716         links.Delete(oldLinkIndex);
717     }
718     return newLinkIndex;
719 }
720
721 public static ILinks<TLink>
722     ↪ DecorateWithAutomaticUniquenessAndUsagesResolution<TLink>(this ILinks<TLink> links)
723 {
724     links = new LinksCascadeUsagesResolver<TLink>(links);
725     links = new NonNullContentsLinkDeletionResolver<TLink>(links);
726     links = new LinksCascadeUniquenessAndUsagesResolver<TLink>(links);
727     return links;
728 }
729 }
730 }

```


./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
15         public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) : base(links) =>
16             ↳ _unaryOne = unaryOne;
17
18         public TLink Increment(TLink unaryNumber)
19         {
20             if (_equalityComparer.Equals(unaryNumber, _unaryOne))
21             {
22                 return Links.GetOrCreate(_unaryOne, _unaryOne);
23             }
24             var source = Links.GetSource(unaryNumber);
25             var target = Links.GetTarget(unaryNumber);
26             if (_equalityComparer.Equals(source, target))
27             {
28                 return Links.GetOrCreate(unaryNumber, _unaryOne);
29             }
30             else
31             {
32                 return Links.GetOrCreate(source, Increment(target));
33             }
34         }
35     }
36 }
```

./Platform.Data.Doublets/ISynchronizedLinks.cs

```
1 using Platform.Data.Constants;
2
```

```

3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets
6  {
7      public interface ISynchronizedLinks<TLink> : ISynchronizedLinks<TLink, ILinks<TLink>,
8          ↳ LinksCombinedConstants<TLink, TLink, int>>, ILinks<TLink>
9      {
10     }
11 }
12
13 ./Platform.Data.Doublets/Link.cs
14
15 using System;
16 using System.Collections;
17 using System.Collections.Generic;
18 using Platform.Exceptions;
19 using Platform.Ranges;
20 using Platform.Singletons;
21 using Platform.Collections.Lists;
22 using Platform.Data.Constants;
23
24 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
25
26 namespace Platform.Data.Doublets
27 {
28     /// <summary>
29     /// Структура описывающая уникальную связь.
30     /// </summary>
31     public struct Link<TLink> : IEquatable<Link<TLink>>, IReadOnlyList<TLink>, IList<TLink>
32     {
33         public static readonly Link<TLink> Null = new Link<TLink>();
34
35         private static readonly LinksCombinedConstants<bool, TLink, int> _constants =
36             ↳ Default<LinksCombinedConstants<bool, TLink, int>>.Instance;
37         private static readonly EqualityComparer<TLink> _equalityComparer =
38             ↳ EqualityComparer<TLink>.Default;
39
40         private const int Length = 3;
41
42         public readonly TLink Index;
43         public readonly TLink Source;
44         public readonly TLink Target;
45
46         public Link(params TLink[] values)
47         {
48             Index = values.Length > _constants.IndexPart ? values[_constants.IndexPart] :
49                 ↳ _constants.Null;
50             Source = values.Length > _constants.SourcePart ? values[_constants.SourcePart] :
51                 ↳ _constants.Null;
52             Target = values.Length > _constants.TargetPart ? values[_constants.TargetPart] :
53                 ↳ _constants.Null;
54         }
55
56         public Link(IList<TLink> values)
57         {
58             Index = values.Count > _constants.IndexPart ? values[_constants.IndexPart] :
59                 ↳ _constants.Null;
60             Source = values.Count > _constants.SourcePart ? values[_constants.SourcePart] :
61                 ↳ _constants.Null;
62             Target = values.Count > _constants.TargetPart ? values[_constants.TargetPart] :
63                 ↳ _constants.Null;
64         }
65
66         public Link(TLink index, TLink source, TLink target)
67         {
68             Index = index;
69             Source = source;
70             Target = target;
71         }
72
73         public Link(TLink source, TLink target)
74             : this(_constants.Null, source, target)
75         {
76             Source = source;
77             Target = target;
78         }
79
80         public static Link<TLink> Create(TLink source, TLink target) => new Link<TLink>(source,
81             ↳ target);
82
83         public override int GetHashCode() => (Index, Source, Target).GetHashCode();
84     }
85 }

```

```

62     public bool IsNull() => _equalityComparer.Equals(Index, _constants.Null)
63         && _equalityComparer.Equals(Source, _constants.Null)
64         && _equalityComparer.Equals(Target, _constants.Null);
65
66     public override bool Equals(object other) => other is Link<TLink> &&
67         ↪ Equals((Link<TLink>)other);
68
69     public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
70         && _equalityComparer.Equals(Source, other.Source)
71         && _equalityComparer.Equals(Target, other.Target);
72
73     public static string ToString(TLink index, TLink source, TLink target) => $"({index}:
74         ↪ {source}->{target})";
75
76     public static string ToString(TLink source, TLink target) => $"({source}->{target})";
77
78     public static implicit operator TLink[] (Link<TLink> link) => link.ToArray();
79
80     public static implicit operator Link<TLink>(TLink[] linkArray) => new
81         ↪ Link<TLink>(linkArray);
82
83     public override string ToString() => _equalityComparer.Equals(Index, _constants.Null) ?
84         ↪ ToString(Source, Target) : ToString(Index, Source, Target);
85
86     #region IList
87
88     public int Count => Length;
89
90     public bool IsReadOnly => true;
91
92     public TLink this[int index]
93     {
94         get
95         {
96             Ensure.OnDebug.ArgumentInRange(index, new Range<int>(0, Length - 1),
97                 ↪ nameof(index));
98             if (index == _constants.IndexPart)
99             {
100                 return Index;
101             }
102             if (index == _constants.SourcePart)
103             {
104                 return Source;
105             }
106             if (index == _constants.TargetPart)
107             {
108                 return Target;
109             }
110             throw new NotSupportedException(); // Impossible path due to
111                 ↪ Ensure.ArgumentInRange
112         }
113         set => throw new NotSupportedException();
114     }
115
116     IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
117
118     public IEnumerator<TLink> GetEnumerator()
119     {
120         yield return Index;
121         yield return Source;
122         yield return Target;
123     }
124
125     public void Add(TLink item) => throw new NotSupportedException();
126
127     public void Clear() => throw new NotSupportedException();
128
129     public bool Contains(TLink item) => IndexOf(item) >= 0;
130
131     public void CopyTo(TLink[] array, int arrayIndex)
132     {
133         Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
134         Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
135             ↪ nameof(arrayIndex));
136         if (arrayIndex + Length > array.Length)
137         {
138             throw new InvalidOperationException();
139         }
140         array[arrayIndex++] = Index;

```

```

134         array[arrayIndex++] = Source;
135         array[arrayIndex] = Target;
136     }
137
138     public bool Remove(TLink item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
139
140     public int IndexOf(TLink item)
141     {
142         if (_equalityComparer.Equals(Index, item))
143         {
144             return _constants.IndexPart;
145         }
146         if (_equalityComparer.Equals(Source, item))
147         {
148             return _constants.SourcePart;
149         }
150         if (_equalityComparer.Equals(Target, item))
151         {
152             return _constants.TargetPart;
153         }
154         return -1;
155     }
156
157     public void Insert(int index, TLink item) => throw new NotSupportedException();
158
159     public void RemoveAt(int index) => throw new NotSupportedException();
160
161     #endregion
162 }
163 }

```

./Platform.Data.Doublets/LinkExtensions.cs

```

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

```

./Platform.Data.Doublets/LinksOperatorBase.cs

```

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

```

./Platform.Data.Doublets/Numbers/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 sourceAddress)
19         {
20             var number = sourceAddress;
21             var nullConstant = Links.Constants.Null;
22             var one = Integer<TLink>.One;
23             var target = nullConstant;
24             for (int i = 0; !_equalityComparer.Equals(number, default) && i <
                ↳ Type<TLink>.BitsLength; i++)
25             {
26                 if (_equalityComparer.Equals(Arithmetic.And(number, one), one))
27                 {
28                     target = _equalityComparer.Equals(target, nullConstant)
29                         ? _powerOf2ToUnaryNumberConverter.Convert(i)
30                         : Links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
31                 }
32                 number = (Integer<TLink>)((ulong)(Integer<TLink>)number >> 1); // Should be
                    ↳ Bit.ShiftRight(number, 1)
33             }
34             return target;
35         }
36     }
37 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Numbers.Unary
8  {
9     public class LinkToItsFrequencyNumberConveter<TLink> : LinksOperatorBase<TLink>,
        ↳ IConverter<Doublet<TLink>, TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↳ EqualityComparer<TLink>.Default;
12
13         private readonly IPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
14         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
15
16         public LinkToItsFrequencyNumberConveter(
17             ILinks<TLink> links,
18             IPropertyOperator<TLink, TLink> frequencyPropertyOperator,
19             IConverter<TLink> unaryNumberToAddressConverter)
20             : base(links)
21         {
22             _frequencyPropertyOperator = frequencyPropertyOperator;
23             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
24         }
25
26         public TLink Convert(Doublet<TLink> doublet)

```

```

27     {
28         var link = Links.SearchOrDefault(douplet.Source, doublet.Target);
29         if (_equalityComparer.Equals(link, default))
30         {
31             throw new ArgumentException($"Link ({douplet}) not found.", nameof(douplet));
32         }
33         var frequency = _frequencyPropertyOperator.Get(link);
34         if (_equalityComparer.Equals(frequency, default))
35         {
36             return default;
37         }
38         var frequencyNumber = Links.GetSource(frequency);
39         return _unaryNumberToAddressConverter.Convert(frequencyNumber);
40     }
41 }
42 }

```

./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

```

```

20     _unaryOne = unaryOne;
21     InitUnaryToUInt64();
22 }
23
24 private void InitUnaryToUInt64()
25 {
26     var one = Integer<TLink>.One;
27     _unaryToUInt64 = new Dictionary<TLink, TLink>
28     {
29         { _unaryOne, one }
30     };
31     var unary = _unaryOne;
32     var number = one;
33     for (var i = 1; i < 64; i++)
34     {
35         unary = Links.GetOrCreate(unary, unary);
36         number = Double(number);
37         _unaryToUInt64.Add(unary, number);
38     }
39 }
40
41 public TLink Convert(TLink unaryNumber)
42 {
43     if (_equalityComparer.Equals(unaryNumber, default))
44     {
45         return default;
46     }
47     if (_equalityComparer.Equals(unaryNumber, _unaryOne))
48     {
49         return Integer<TLink>.One;
50     }
51     var source = Links.GetSource(unaryNumber);
52     var target = Links.GetTarget(unaryNumber);
53     if (_equalityComparer.Equals(source, target))
54     {
55         return _unaryToUInt64[unaryNumber];
56     }
57     else
58     {
59         var result = _unaryToUInt64[source];
60         TLink lastValue;
61         while (!_unaryToUInt64.TryGetValue(target, out lastValue))
62         {
63             source = Links.GetSource(target);
64             result = Arithmetic<TLink>.Add(result, _unaryToUInt64[source]);
65             target = Links.GetTarget(target);
66         }
67         result = Arithmetic<TLink>.Add(result, lastValue);
68         return result;
69     }
70 }
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 private static TLink Double(TLink number) => (Integer<TLink>)((Integer<TLink>)number *
74     ↪ 2UL);
75 }

```

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

```

1 using System.Collections.Generic;
2 using Platform.Interfaces;
3 using Platform.Reflection;
4 using Platform.Numbers;
5 using System.Runtime.CompilerServices;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Data.Doublets.Numbers.Unary
10 {
11     public class UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
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)

```

```

19     {
20         _unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
21         for (int i = 0; i < Type<TLink>.BitsLength; i++)
22         {
23             _unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
24         }
25     }
26
27     public TLink Convert(TLink sourceNumber)
28     {
29         var nullConstant = Links.Constants.Null;
30         var source = sourceNumber;
31         var target = nullConstant;
32         if (!_equalityComparer.Equals(source, nullConstant))
33         {
34             while (true)
35             {
36                 if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
37                 {
38                     SetBit(ref target, powerOf2Index);
39                     break;
40                 }
41                 else
42                 {
43                     powerOf2Index = _unaryNumberPowerOf2Indicies[Links.GetSource(source)];
44                     SetBit(ref target, powerOf2Index);
45                     source = Links.GetTarget(source);
46                 }
47             }
48         }
49         return target;
50     }
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     private static void SetBit(ref TLink target, int powerOf2Index) => target =
54     ↪ (Integer<TLink>)((Integer<TLink>)target | 1UL << powerOf2Index); // Should be
55     ↪ Math.Or(target, Math.ShiftLeft(One, powerOf2Index))
56 }

```

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

```

1  using System.Linq;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.PropertyOperators
8  {
9      public class PropertiesOperator<TLink> : LinksOperatorBase<TLink>,
10     ↪ IPropertiesOperator<TLink, TLink, TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13         ↪ EqualityComparer<TLink>.Default;
14
15         public PropertiesOperator(ILinks<TLink> links) : base(links) { }
16
17         public TLink GetValue(TLink @object, TLink property)
18         {
19             var objectProperty = Links.SearchOrDefault(@object, property);
20             if (_equalityComparer.Equals(objectProperty, default))
21             {
22                 return default;
23             }
24             var valueLink = Links.All(Links.Constants.Any, objectProperty).SingleOrDefault();
25             if (valueLink == null)
26             {
27                 return default;
28             }
29             return Links.GetTarget(valueLink[Links.Constants.IndexPart]);
30         }
31
32         public void SetValue(TLink @object, TLink property, TLink value)
33         {
34             var objectProperty = Links.GetOrCreate(@object, property);
35             Links.DeleteMany(Links.AllIndices(Links.Constants.Any, objectProperty));
36             Links.GetOrCreate(objectProperty, value);
37         }
38     }
39 }

```



```
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         private TLink GetValue(TLink container) => _equalityComparer.Equals(container, default)
59             ↪ ? default : Links.GetTarget(container);
60
61         public void Set(TLink link, TLink value)
62         {
63             var property = Links.GetOrCreate(link, _propertyMarker);
64             var container = GetContainer(property);
65             if (_equalityComparer.Equals(container, default))
66             {
67                 Links.GetOrCreate(property, value);
68             }
69             else
70             {
71                 Links.Update(container, property, value);
72             }
73         }
74     }
75 }
```

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

```
1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using System.Runtime.InteropServices;
5 using Platform.Disposables;
6 using Platform.Singletons;
7 using Platform.Collections.Arrays;
8 using Platform.Numbers;
9 using Platform.Unsafe;
10 using Platform.Memory;
11 using Platform.Data.Exceptions;
12 using Platform.Data.Constants;
13 using static Platform.Numbers.Arithmetic;
14
15 #pragma warning disable 0649
16 #pragma warning disable 169
17 #pragma warning disable 618
18 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
19
20 // ReSharper disable StaticMemberInGenericType
21 // ReSharper disable BuiltInTypeReferenceStyle
22 // ReSharper disable MemberCanBePrivate.Local
23 // ReSharper disable UnusedMember.Local
24
25 namespace Platform.Data.Doublets.ResizableDirectMemory
26 {
27     public partial class ResizableDirectMemoryLinks<TLink> : DisposableBase, ILinks<TLink>
28     {
29         private static readonly EqualityComparer<TLink> _equalityComparer =
30             ↳ EqualityComparer<TLink>.Default;
31         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
32
33         /// <summary>Возвращает размер одной связи в байтах.</summary>
34         public static readonly int LinkSizeInBytes = Structure<Link>.Size;
35
36         public static readonly int LinkHeaderSizeInBytes = Structure<LinksHeader>.Size;
37
38         public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
39
40         private struct Link
41         {
42             public static readonly int SourceOffset = Marshal.OffsetOf(typeof(Link),
43                 ↳ nameof(Source)).ToInt32();
44             public static readonly int TargetOffset = Marshal.OffsetOf(typeof(Link),
45                 ↳ nameof(Target)).ToInt32();
46             public static readonly int LeftAsSourceOffset = Marshal.OffsetOf(typeof(Link),
47                 ↳ nameof(LeftAsSource)).ToInt32();
48             public static readonly int RightAsSourceOffset = Marshal.OffsetOf(typeof(Link),
49                 ↳ nameof(RightAsSource)).ToInt32();
50             public static readonly int SizeAsSourceOffset = Marshal.OffsetOf(typeof(Link),
51                 ↳ nameof(SizeAsSource)).ToInt32();
52             public static readonly int LeftAsTargetOffset = Marshal.OffsetOf(typeof(Link),
53                 ↳ nameof(LeftAsTarget)).ToInt32();
54             public static readonly int RightAsTargetOffset = Marshal.OffsetOf(typeof(Link),
55                 ↳ nameof(RightAsTarget)).ToInt32();
56             public static readonly int SizeAsTargetOffset = Marshal.OffsetOf(typeof(Link),
57                 ↳ nameof(SizeAsTarget)).ToInt32();
58
59             public TLink Source;
60             public TLink Target;
61             public TLink LeftAsSource;
62             public TLink RightAsSource;
63             public TLink SizeAsSource;
64             public TLink LeftAsTarget;
65             public TLink RightAsTarget;
66             public TLink SizeAsTarget;
67
68             [MethodImpl(MethodImplOptions.AggressiveInlining)]
69             public static TLink GetSource(IntPtr pointer) => (pointer +
70                 ↳ SourceOffset).GetValue<TLink>();
71             [MethodImpl(MethodImplOptions.AggressiveInlining)]
72             public static TLink GetTarget(IntPtr pointer) => (pointer +
73                 ↳ TargetOffset).GetValue<TLink>();
74             [MethodImpl(MethodImplOptions.AggressiveInlining)]
75             public static TLink GetLeftAsSource(IntPtr pointer) => (pointer +
76                 ↳ LeftAsSourceOffset).GetValue<TLink>();
77             [MethodImpl(MethodImplOptions.AggressiveInlining)]
78             public static TLink GetRightAsSource(IntPtr pointer) => (pointer +
79                 ↳ RightAsSourceOffset).GetValue<TLink>();
80         }
81     }
82 }
```

```

67     [MethodImpl(MethodImplOptions.AggressiveInlining)]
68     public static TLink GetSizeAsSource(IntPtr pointer) => (pointer +
    ↳ SizeAsSourceOffset).GetValue<TLink>();
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     public static TLink GetLeftAsTarget(IntPtr pointer) => (pointer +
    ↳ LeftAsTargetOffset).GetValue<TLink>();
71     [MethodImpl(MethodImplOptions.AggressiveInlining)]
72     public static TLink GetRightAsTarget(IntPtr pointer) => (pointer +
    ↳ RightAsTargetOffset).GetValue<TLink>();
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     public static TLink GetSizeAsTarget(IntPtr pointer) => (pointer +
    ↳ SizeAsTargetOffset).GetValue<TLink>();
75
76     [MethodImpl(MethodImplOptions.AggressiveInlining)]
77     public static void SetSource(IntPtr pointer, TLink value) => (pointer +
    ↳ SourceOffset).SetValue(value);
78     [MethodImpl(MethodImplOptions.AggressiveInlining)]
79     public static void SetTarget(IntPtr pointer, TLink value) => (pointer +
    ↳ TargetOffset).SetValue(value);
80     [MethodImpl(MethodImplOptions.AggressiveInlining)]
81     public static void SetLeftAsSource(IntPtr pointer, TLink value) => (pointer +
    ↳ LeftAsSourceOffset).SetValue(value);
82     [MethodImpl(MethodImplOptions.AggressiveInlining)]
83     public static void SetRightAsSource(IntPtr pointer, TLink value) => (pointer +
    ↳ RightAsSourceOffset).SetValue(value);
84     [MethodImpl(MethodImplOptions.AggressiveInlining)]
85     public static void SetSizeAsSource(IntPtr pointer, TLink value) => (pointer +
    ↳ SizeAsSourceOffset).SetValue(value);
86     [MethodImpl(MethodImplOptions.AggressiveInlining)]
87     public static void SetLeftAsTarget(IntPtr pointer, TLink value) => (pointer +
    ↳ LeftAsTargetOffset).SetValue(value);
88     [MethodImpl(MethodImplOptions.AggressiveInlining)]
89     public static void SetRightAsTarget(IntPtr pointer, TLink value) => (pointer +
    ↳ RightAsTargetOffset).SetValue(value);
90     [MethodImpl(MethodImplOptions.AggressiveInlining)]
91     public static void SetSizeAsTarget(IntPtr pointer, TLink value) => (pointer +
    ↳ SizeAsTargetOffset).SetValue(value);
92 }
93
94 private struct LinksHeader
95 {
96     public static readonly int AllocatedLinksOffset =
    ↳ Marshal.OffsetOf(typeof(LinksHeader), nameof(AllocatedLinks)).ToInt32();
97     public static readonly int ReservedLinksOffset =
    ↳ Marshal.OffsetOf(typeof(LinksHeader), nameof(ReservedLinks)).ToInt32();
98     public static readonly int FreeLinksOffset = Marshal.OffsetOf(typeof(LinksHeader),
    ↳ nameof(FreeLinks)).ToInt32();
99     public static readonly int FirstFreeLinkOffset =
    ↳ Marshal.OffsetOf(typeof(LinksHeader), nameof(FirstFreeLink)).ToInt32();
100    public static readonly int FirstAsSourceOffset =
    ↳ Marshal.OffsetOf(typeof(LinksHeader), nameof(FirstAsSource)).ToInt32();
101    public static readonly int FirstAsTargetOffset =
    ↳ Marshal.OffsetOf(typeof(LinksHeader), nameof(FirstAsTarget)).ToInt32();
102    public static readonly int LastFreeLinkOffset =
    ↳ Marshal.OffsetOf(typeof(LinksHeader), nameof(LastFreeLink)).ToInt32();
103
104    public TLink AllocatedLinks;
105    public TLink ReservedLinks;
106    public TLink FreeLinks;
107    public TLink FirstFreeLink;
108    public TLink FirstAsSource;
109    public TLink FirstAsTarget;
110    public TLink LastFreeLink;
111    public TLink Reserved8;
112
113    [MethodImpl(MethodImplOptions.AggressiveInlining)]
114    public static TLink GetAllocatedLinks(IntPtr pointer) => (pointer +
    ↳ AllocatedLinksOffset).GetValue<TLink>();
115    [MethodImpl(MethodImplOptions.AggressiveInlining)]
116    public static TLink GetReservedLinks(IntPtr pointer) => (pointer +
    ↳ ReservedLinksOffset).GetValue<TLink>();
117    [MethodImpl(MethodImplOptions.AggressiveInlining)]
118    public static TLink GetFreeLinks(IntPtr pointer) => (pointer +
    ↳ FreeLinksOffset).GetValue<TLink>();
119    [MethodImpl(MethodImplOptions.AggressiveInlining)]
120    public static TLink GetFirstFreeLink(IntPtr pointer) => (pointer +
    ↳ FirstFreeLinkOffset).GetValue<TLink>();

```

```

121     [MethodImpl(MethodImplOptions.AggressiveInlining)]
122     public static TLink GetFirstAsSource(IntPtr pointer) => (pointer +
    ↪ FirstAsSourceOffset).GetValue<TLink>();
123     [MethodImpl(MethodImplOptions.AggressiveInlining)]
124     public static TLink GetFirstAsTarget(IntPtr pointer) => (pointer +
    ↪ FirstAsTargetOffset).GetValue<TLink>();
125     [MethodImpl(MethodImplOptions.AggressiveInlining)]
126     public static TLink GetLastFreeLink(IntPtr pointer) => (pointer +
    ↪ LastFreeLinkOffset).GetValue<TLink>();
127
128     [MethodImpl(MethodImplOptions.AggressiveInlining)]
129     public static IntPtr GetFirstAsSourcePointer(IntPtr pointer) => pointer +
    ↪ FirstAsSourceOffset;
130     [MethodImpl(MethodImplOptions.AggressiveInlining)]
131     public static IntPtr GetFirstAsTargetPointer(IntPtr pointer) => pointer +
    ↪ FirstAsTargetOffset;
132
133     [MethodImpl(MethodImplOptions.AggressiveInlining)]
134     public static void SetAllocatedLinks(IntPtr pointer, TLink value) => (pointer +
    ↪ AllocatedLinksOffset).SetValue(value);
135     [MethodImpl(MethodImplOptions.AggressiveInlining)]
136     public static void SetReservedLinks(IntPtr pointer, TLink value) => (pointer +
    ↪ ReservedLinksOffset).SetValue(value);
137     [MethodImpl(MethodImplOptions.AggressiveInlining)]
138     public static void SetFreeLinks(IntPtr pointer, TLink value) => (pointer +
    ↪ FreeLinksOffset).SetValue(value);
139     [MethodImpl(MethodImplOptions.AggressiveInlining)]
140     public static void SetFirstFreeLink(IntPtr pointer, TLink value) => (pointer +
    ↪ FirstFreeLinkOffset).SetValue(value);
141     [MethodImpl(MethodImplOptions.AggressiveInlining)]
142     public static void SetFirstAsSource(IntPtr pointer, TLink value) => (pointer +
    ↪ FirstAsSourceOffset).SetValue(value);
143     [MethodImpl(MethodImplOptions.AggressiveInlining)]
144     public static void SetFirstAsTarget(IntPtr pointer, TLink value) => (pointer +
    ↪ FirstAsTargetOffset).SetValue(value);
145     [MethodImpl(MethodImplOptions.AggressiveInlining)]
146     public static void SetLastFreeLink(IntPtr pointer, TLink value) => (pointer +
    ↪ LastFreeLinkOffset).SetValue(value);
147 }
148
149 private readonly long _memoryReservationStep;
150
151 private readonly IResizableDirectMemory _memory;
152 private IntPtr _header;
153 private IntPtr _links;
154
155 private LinksTargetsTreeMethods _targetsTreeMethods;
156 private LinksSourcesTreeMethods _sourcesTreeMethods;
157
158 // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
    ↪ нужно использовать не список а дерево, так как так можно быстрее проверить на
    ↪ наличие связи внутри
159 private UnusedLinksListMethods _unusedLinksListMethods;
160
161 /// <summary>
162 /// Возвращает общее число связей находящихся в хранилище.
163 /// </summary>
164 private TLink Total => Subtract(LinksHeader.GetAllocatedLinks(_header),
    ↪ LinksHeader.GetFreeLinks(_header));
165
166 public LinksCombinedConstants<TLink, TLink, int> Constants { get; }
167
168 public ResizableDirectMemoryLinks(string address)
169     : this(address, DefaultLinksSizeStep)
170 {
171 }
172
173 /// <summary>
174 /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
    ↪ минимальным шагом расширения базы данных.
175 /// </summary>
176 /// <param name="address">Полный путь к файлу базы данных.</param>
177 /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
    ↪ байтах.</param>
178 public ResizableDirectMemoryLinks(string address, long memoryReservationStep)
179     : this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
    ↪ memoryReservationStep)
180 {

```

```

181     }
182
183     public ResizableDirectMemoryLinks(IResizableDirectMemory memory)
184         : this(memory, DefaultLinksSizeStep)
185     {
186     }
187
188     public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
189     ↪ memoryReservationStep)
189     {
190         Constants = Default<LinksCombinedConstants<TLink, TLink, int>>.Instance;
191         _memory = memory;
192         _memoryReservationStep = memoryReservationStep;
193         if (memory.ReservedCapacity < memoryReservationStep)
194         {
195             memory.ReservedCapacity = memoryReservationStep;
196         }
197         SetPointers(_memory);
198         // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
199         _memory.UsedCapacity = ((long)(Integer<TLink>)LinksHeader.GetAllocatedLinks(_header)
200         ↪ * LinkSizeInBytes) + LinkHeaderSizeInBytes;
201         // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
202         LinksHeader.SetReservedLinks(_header, (Integer<TLink>)((_memory.ReservedCapacity -
203         ↪ LinkHeaderSizeInBytes) / LinkSizeInBytes));
204     }
205
206     [MethodImpl(MethodImplOptions.AggressiveInlining)]
207     public TLink Count(IList<TLink> restrictions)
208     {
209         // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
210         if (restrictions.Count == 0)
211         {
212             return Total;
213         }
214         if (restrictions.Count == 1)
215         {
216             var index = restrictions[Constants.IndexPart];
217             if (_equalityComparer.Equals(index, Constants.Any))
218             {
219                 return Total;
220             }
221             return Exists(index) ? Integer<TLink>.One : Integer<TLink>.Zero;
222         }
223         if (restrictions.Count == 2)
224         {
225             var index = restrictions[Constants.IndexPart];
226             var value = restrictions[1];
227             if (_equalityComparer.Equals(index, Constants.Any))
228             {
229                 if (_equalityComparer.Equals(value, Constants.Any))
230                 {
231                     return Total; // Any - как отсутствие ограничения
232                 }
233                 return Add(_sourcesTreeMethods.CountUsages(value),
234                 ↪ _targetsTreeMethods.CountUsages(value));
235             }
236             else
237             {
238                 if (!Exists(index))
239                 {
240                     return Integer<TLink>.Zero;
241                 }
242                 if (_equalityComparer.Equals(value, Constants.Any))
243                 {
244                     return Integer<TLink>.One;
245                 }
246                 var storedLinkValue = GetLinkUnsafe(index);
247                 if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
248                 ↪ _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
249                 {
250                     return Integer<TLink>.One;
251                 }
252                 return Integer<TLink>.Zero;
253             }
254         }
255         if (restrictions.Count == 3)
256         {
257             var index = restrictions[Constants.IndexPart];

```

```

255 var source = restrictions[Constants.SourcePart];
256 var target = restrictions[Constants.TargetPart];
257
258 if (_equalityComparer.Equals(index, Constants.Any))
259 {
260     if (_equalityComparer.Equals(source, Constants.Any) &&
261         ↪ _equalityComparer.Equals(target, Constants.Any))
262     {
263         return Total;
264     }
265     else if (_equalityComparer.Equals(source, Constants.Any))
266     {
267         return _targetsTreeMethods.CountUsages(target);
268     }
269     else if (_equalityComparer.Equals(target, Constants.Any))
270     {
271         return _sourcesTreeMethods.CountUsages(source);
272     }
273     else //if(source != Any && target != Any)
274     {
275         // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
276         var link = _sourcesTreeMethods.Search(source, target);
277         return _equalityComparer.Equals(link, Constants.Null) ?
278             ↪ Integer<TLink>.Zero : Integer<TLink>.One;
279     }
280 }
281 else
282 {
283     if (!Exists(index))
284     {
285         return Integer<TLink>.Zero;
286     }
287     if (_equalityComparer.Equals(source, Constants.Any) &&
288         ↪ _equalityComparer.Equals(target, Constants.Any))
289     {
290         return Integer<TLink>.One;
291     }
292     var storedLinkValue = GetLinkUnsafe(index);
293     if (!_equalityComparer.Equals(source, Constants.Any) &&
294         ↪ !_equalityComparer.Equals(target, Constants.Any))
295     {
296         if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source) &&
297             ↪ _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
298         {
299             return Integer<TLink>.One;
300         }
301         return Integer<TLink>.Zero;
302     }
303     var value = default(TLink);
304     if (_equalityComparer.Equals(source, Constants.Any))
305     {
306         value = target;
307     }
308     if (_equalityComparer.Equals(target, Constants.Any))
309     {
310         value = source;
311     }
312     if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
313         ↪ _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
314     {
315         return Integer<TLink>.One;
316     }
317     return Integer<TLink>.Zero;
318 }
319 }
320 throw new NotSupportedException("Другие размеры и способы ограничений не
321     ↪ поддерживаются.");
322 }
323
324 [MethodImpl(MethodImplOptions.AggressiveInlining)]
325 public TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
326 {
327     if (restrictions.Count == 0)
328     {
329     }
330     for (TLink link = Integer<TLink>.One; _comparer.Compare(link,
331         ↪ (Integer<TLink>)LinksHeader.GetAllocatedLinks(_header)) <= 0; link =
332         ↪ Increment(link))
333     {

```

```

326         if (Exists(link) && _equalityComparer.Equals(handler(GetLinkStruct(link)),
327             ↪ Constants.Break))
328         {
329             return Constants.Break;
330         }
331     }
332     return Constants.Continue;
333 }
334 if (restrictions.Count == 1)
335 {
336     var index = restrictions[Constants.IndexPart];
337     if (_equalityComparer.Equals(index, Constants.Any))
338     {
339         return Each(handler, ArrayPool<TLink>.Empty);
340     }
341     if (!Exists(index))
342     {
343         return Constants.Continue;
344     }
345     return handler(GetLinkStruct(index));
346 }
347 if (restrictions.Count == 2)
348 {
349     var index = restrictions[Constants.IndexPart];
350     var value = restrictions[1];
351     if (_equalityComparer.Equals(index, Constants.Any))
352     {
353         if (_equalityComparer.Equals(value, Constants.Any))
354         {
355             return Each(handler, ArrayPool<TLink>.Empty);
356         }
357         if (_equalityComparer.Equals(Each(handler, new[] { index, value,
358             ↪ Constants.Any }), Constants.Break))
359         {
360             return Constants.Break;
361         }
362         return Each(handler, new[] { index, Constants.Any, value });
363     }
364     else
365     {
366         if (!Exists(index))
367         {
368             return Constants.Continue;
369         }
370         if (_equalityComparer.Equals(value, Constants.Any))
371         {
372             return handler(GetLinkStruct(index));
373         }
374         var storedLinkValue = GetLinkUnsafe(index);
375         if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
376             ↪ _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
377         {
378             return handler(GetLinkStruct(index));
379         }
380         return Constants.Continue;
381     }
382 }
383 if (restrictions.Count == 3)
384 {
385     var index = restrictions[Constants.IndexPart];
386     var source = restrictions[Constants.SourcePart];
387     var target = restrictions[Constants.TargetPart];
388     if (_equalityComparer.Equals(index, Constants.Any))
389     {
390         if (_equalityComparer.Equals(source, Constants.Any) &&
391             ↪ _equalityComparer.Equals(target, Constants.Any))
392         {
393             return Each(handler, ArrayPool<TLink>.Empty);
394         }
395         else if (_equalityComparer.Equals(source, Constants.Any))
396         {
397             return _targetsTreeMethods.EachUsage(target, handler);
398         }
399         else if (_equalityComparer.Equals(target, Constants.Any))
400         {
399             return _sourcesTreeMethods.EachUsage(source, handler);
400         }

```

```

401     else //if(source != Any && target != Any)
402     {
403         var link = _sourcesTreeMethods.Search(source, target);
404         return _equalityComparer.Equals(link, Constants.Null) ?
            ↳ Constants.Continue : handler(GetLinkStruct(link));
405     }
406 }
407 else
408 {
409     if (!Exists(index))
410     {
411         return Constants.Continue;
412     }
413     if (_equalityComparer.Equals(source, Constants.Any) &&
        ↳ _equalityComparer.Equals(target, Constants.Any))
414     {
415         return handler(GetLinkStruct(index));
416     }
417     var storedLinkValue = GetLinkUnsafe(index);
418     if (!_equalityComparer.Equals(source, Constants.Any) &&
        ↳ !_equalityComparer.Equals(target, Constants.Any))
419     {
420         if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source) &&
            ↳ _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
421         {
422             return handler(GetLinkStruct(index));
423         }
424         return Constants.Continue;
425     }
426     var value = default(TLink);
427     if (_equalityComparer.Equals(source, Constants.Any))
428     {
429         value = target;
430     }
431     if (_equalityComparer.Equals(target, Constants.Any))
432     {
433         value = source;
434     }
435     if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
        ↳ _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
436     {
437         return handler(GetLinkStruct(index));
438     }
439     return Constants.Continue;
440 }
441 }
442 }
443 }
444 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↳ поддерживаются.");
445 }
446
447 /// <remarks>
448 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
    ↳ в другом месте (но не в менеджере памяти, а в логике Links)
449 /// </remarks>
450 [MethodImpl(MethodImplOptions.AggressiveInlining)]
451 public TLink Update(IList<TLink> values)
452 {
453     var linkIndex = values[Constants.IndexPart];
454     var link = GetLinkUnsafe(linkIndex);
455     // Будет корректно работать только в том случае, если пространство выделенной связи
    ↳ предварительно заполнено нулями
456     if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
457     {
458         _sourcesTreeMethods.Detach(LinksHeader.GetFirstAsSourcePointer(_header),
            ↳ linkIndex);
459     }
460     if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
461     {
462         _targetsTreeMethods.Detach(LinksHeader.GetFirstAsTargetPointer(_header),
            ↳ linkIndex);
463     }
464     Link.SetSource(link, values[Constants.SourcePart]);
465     Link.SetTarget(link, values[Constants.TargetPart]);
466     if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
467     {
468         _sourcesTreeMethods.Attach(LinksHeader.GetFirstAsSourcePointer(_header),
            ↳ linkIndex);

```



```

469     }
470     if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
471     {
472         _targetsTreeMethods.Attach(LinksHeader.GetFirstAsTargetPointer(_header),
473             ↪ linkIndex);
474     }
475     return linkIndex;
476 }
477 [MethodImpl(MethodImplOptions.AggressiveInlining)]
478 public Link<TLink> GetLinkStruct(TLink linkIndex)
479 {
480     var link = GetLinkUnsafe(linkIndex);
481     return new Link<TLink>(linkIndex, Link.GetSource(link), Link.GetTarget(link));
482 }
483
484 [MethodImpl(MethodImplOptions.AggressiveInlining)]
485 private IntPtr GetLinkUnsafe(TLink linkIndex) => _links.GetElement(LinkSizeInBytes,
486     ↪ linkIndex);
487
488 /// <remarks>
489 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
490 ↪ пространство
491 /// </remarks>
492 public TLink Create()
493 {
494     var freeLink = LinksHeader.GetFirstFreeLink(_header);
495     if (!_equalityComparer.Equals(freeLink, Constants.Null))
496     {
497         _unusedLinksListMethods.Detach(freeLink);
498     }
499     else
500     {
501         if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
502             ↪ Constants.MaxPossibleIndex) > 0)
503         {
504             throw new
505                 ↪ LinksLimitReachedException((Integer<TLink>)Constants.MaxPossibleIndex);
506         }
507         if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
508             ↪ Decrement(LinksHeader.GetReservedLinks(_header))) >= 0)
509         {
510             _memory.ReservedCapacity += _memory.ReservationStep;
511             SetPointers(_memory);
512             LinksHeader.SetReservedLinks(_header,
513                 ↪ (Integer<TLink>)(_memory.ReservedCapacity / LinkSizeInBytes));
514         }
515         LinksHeader.SetAllocatedLinks(_header,
516             ↪ Increment(LinksHeader.GetAllocatedLinks(_header)));
517         _memory.UsedCapacity += LinkSizeInBytes;
518         freeLink = LinksHeader.GetAllocatedLinks(_header);
519     }
520     return freeLink;
521 }
522
523 public void Delete(TLink link)
524 {
525     if (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) < 0)
526     {
527         _unusedLinksListMethods.AttachAsFirst(link);
528     }
529     else if (_equalityComparer.Equals(link, LinksHeader.GetAllocatedLinks(_header)))
530     {
531         LinksHeader.SetAllocatedLinks(_header,
532             ↪ Decrement(LinksHeader.GetAllocatedLinks(_header)));
533         _memory.UsedCapacity -= LinkSizeInBytes;
534         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
535         ↪ пока не дойдём до первой существующей связи
536         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
537         while ((_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
538             ↪ Integer<TLink>.Zero) > 0) &&
539             ↪ IsUnusedLink(LinksHeader.GetAllocatedLinks(_header)))
540         {
541             _unusedLinksListMethods.Detach(LinksHeader.GetAllocatedLinks(_header));
542             LinksHeader.SetAllocatedLinks(_header,
543                 ↪ Decrement(LinksHeader.GetAllocatedLinks(_header)));
544             _memory.UsedCapacity -= LinkSizeInBytes;
545         }
546     }
547 }

```

```

534     }
535 }
536
537 /// <remarks>
538 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
539 /// ↪ адрес реально поменялся
540 ///
541 /// Указатель this.links может быть в том же месте,
542 /// так как 0-я связь не используется и имеет такой же размер как Header,
543 /// поэтому header размещается в том же месте, что и 0-я связь
544 /// </remarks>
545 private void SetPointers(IDirectMemory memory)
546 {
547     if (memory == null)
548     {
549         _links = IntPtr.Zero;
550         _header = _links;
551         _unusedLinksListMethods = null;
552         _targetsTreeMethods = null;
553         _unusedLinksListMethods = null;
554     }
555     else
556     {
557         _links = memory.Pointer;
558         _header = _links;
559         _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
560         _targetsTreeMethods = new LinksTargetsTreeMethods(this);
561         _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
562     }
563 }
564
565 [MethodImpl(MethodImplOptions.AggressiveInlining)]
566 private bool Exists(TLink link)
567 => (_comparer.Compare(link, Constants.MinPossibleIndex) >= 0)
568     && (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) <= 0)
569     && !IsUnusedLink(link);
570
571 [MethodImpl(MethodImplOptions.AggressiveInlining)]
572 private bool IsUnusedLink(TLink link)
573 => _equalityComparer.Equals(LinksHeader.GetFirstFreeLink(_header), link)
574     || (_equalityComparer.Equals(Link.GetSizeAsSource(GetLinkUnsafe(link)),
575     ↪ Constants.Null)
576     && !_equalityComparer.Equals(Link.GetSource(GetLinkUnsafe(link)), Constants.Null));
577
578 #region DisposableBase
579
580 protected override bool AllowMultipleDisposeCalls => true;
581
582 protected override void Dispose(bool manual, bool wasDisposed)
583 {
584     if (!wasDisposed)
585     {
586         SetPointers(null);
587         _memory.DisposeIfPossible();
588     }
589 }
590
591 #endregion
592 }
593

```

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

```

1  using System;
2  using Platform.Unsafe;
3  using Platform.Collections.Methods.Lists;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.ResizableDirectMemory
8  {
9      partial class ResizableDirectMemoryLinks<TLink>
10     {
11         private class UnusedLinksListMethods : CircularDoublyLinkedListMethods<TLink>
12         {
13             private readonly IntPtr _links;
14             private readonly IntPtr _header;
15
16             public UnusedLinksListMethods(IntPtr links, IntPtr header)
17             {
18                 _links = links;

```

```

19         _header = header;
20     }
21
22     protected override TLink GetFirst() => (_header +
23     ↪ LinksHeader.FirstFreeLinkOffset).GetValue<TLink>();
24
25     protected override TLink GetLast() => (_header +
26     ↪ LinksHeader.LastFreeLinkOffset).GetValue<TLink>();
27
28     protected override TLink GetPrevious(TLink element) =>
29     ↪ (_links.GetElement(LinkSizeInBytes, element) +
30     ↪ Link.SourceOffset).GetValue<TLink>();
31
32     protected override TLink GetNext(TLink element) =>
33     ↪ (_links.GetElement(LinkSizeInBytes, element) +
34     ↪ Link.TargetOffset).GetValue<TLink>();
35
36     protected override TLink GetSize() => (_header +
37     ↪ LinksHeader.FreeLinksOffset).GetValue<TLink>();
38
39     protected override void SetFirst(TLink element) => (_header +
40     ↪ LinksHeader.FirstFreeLinkOffset).SetValue(element);
41
42     protected override void SetLast(TLink element) => (_header +
43     ↪ LinksHeader.LastFreeLinkOffset).SetValue(element);
44
45     protected override void SetPrevious(TLink element, TLink previous) =>
46     ↪ (_links.GetElement(LinkSizeInBytes, element) +
47     ↪ Link.SourceOffset).SetValue(previous);
48
49     protected override void SetNext(TLink element, TLink next) =>
50     ↪ (_links.GetElement(LinkSizeInBytes, element) + Link.TargetOffset).SetValue(next);
51
52     protected override void SetSize(TLink size) => (_header +
53     ↪ LinksHeader.FreeLinksOffset).SetValue(size);
54 }
55 }
56 }

```

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

```

1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Numbers;
6  using Platform.Unsafe;
7  using Platform.Collections.Methods.Trees;
8  using Platform.Data.Constants;
9
10 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12 namespace Platform.Data.Doublets.ResizableDirectMemory
13 {
14     partial class ResizableDirectMemoryLinks<TLink>
15     {
16         private abstract class LinksTreeMethodsBase :
17         ↪ SizedAndThreadedAVLBalancedTreeMethods<TLink>
18         {
19             private static readonly EqualityComparer<TLink> _equalityComparer =
20             ↪ EqualityComparer<TLink>.Default;
21
22             private readonly ResizableDirectMemoryLinks<TLink> _memory;
23             private readonly LinksCombinedConstants<TLink, TLink, int> _constants;
24             protected readonly IntPtr Links;
25             protected readonly IntPtr Header;
26
27             LinksTreeMethodsBase(ResizableDirectMemoryLinks<TLink> memory)
28             {
29                 Links = memory._links;
30                 Header = memory._header;
31                 _memory = memory;
32                 _constants = memory.Constants;
33             }
34
35             [MethodImpl(MethodImplOptions.AggressiveInlining)]
36             protected abstract TLink GetTreeRoot();
37
38             [MethodImpl(MethodImplOptions.AggressiveInlining)]
39             protected abstract TLink GetBasePartValue(TLink link);
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99

```

```

39 public TLink this[TLink index]
40 {
41     get
42     {
43         var root = GetTreeRoot();
44         if (GreaterOrEqualThan(index, GetSize(root)))
45         {
46             return GetZero();
47         }
48         while (!EqualToZero(root))
49         {
50             var left = GetLeftOrDefault(root);
51             var leftSize = GetSizeOrZero(left);
52             if (LessThan(index, leftSize))
53             {
54                 root = left;
55                 continue;
56             }
57             if (IsEquals(index, leftSize))
58             {
59                 return root;
60             }
61             root = GetRightOrDefault(root);
62             index = Subtract(index, Increment(leftSize));
63         }
64         return GetZero(); // TODO: Impossible situation exception (only if tree
        ↪ structure broken)
65     }
66 }
67
68 // TODO: Return indices range instead of references count
69 public TLink CountUsages(TLink link)
70 {
71     var root = GetTreeRoot();
72     var total = GetSize(root);
73     var totalRightIgnore = GetZero();
74     while (!EqualToZero(root))
75     {
76         var @base = GetBasePartValue(root);
77         if (LessOrEqualThan(@base, link))
78         {
79             root = GetRightOrDefault(root);
80         }
81         else
82         {
83             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
84             root = GetLeftOrDefault(root);
85         }
86     }
87     root = GetTreeRoot();
88     var totalLeftIgnore = GetZero();
89     while (!EqualToZero(root))
90     {
91         var @base = GetBasePartValue(root);
92         if (GreaterOrEqualThan(@base, link))
93         {
94             root = GetLeftOrDefault(root);
95         }
96         else
97         {
98             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
99             root = GetRightOrDefault(root);
100         }
101     }
102     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
103 }
104
105 public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
106 {
107     var root = GetTreeRoot();
108     if (EqualToZero(root))
109     {
110         return _constants.Continue;
111     }
112     TLink first = GetZero(), current = root;
113     while (!EqualToZero(current))
114     {
115

```

```

116     var @base = GetBasePartValue(current);
117     if (GreaterOrEqualThan(@base, link))
118     {
119         if (IsEquals(@base, link))
120         {
121             first = current;
122         }
123         current = GetLeftOrDefault(current);
124     }
125     else
126     {
127         current = GetRightOrDefault(current);
128     }
129 }
130 if (!EqualToZero(first))
131 {
132     current = first;
133     while (true)
134     {
135         if (IsEquals(handler(_memory.GetLinkStruct(current)), _constants.Break))
136         {
137             return _constants.Break;
138         }
139         current = GetNext(current);
140         if (EqualToZero(current) || !IsEquals(GetBasePartValue(current), link))
141         {
142             break;
143         }
144     }
145 }
146 return _constants.Continue;
147 }
148
149 protected override void PrintNodeValue(TLink node, StringBuilder sb)
150 {
151     sb.Append(' ');
152     sb.Append((Links.GetElement(LinkSizeInBytes, node) +
153 ↪ Link.SourceOffset).GetValue<TLink>());
154     sb.Append('-');
155     sb.Append('>');
156     sb.Append((Links.GetElement(LinkSizeInBytes, node) +
157 ↪ Link.TargetOffset).GetValue<TLink>());
158 }
159
160 private class LinksSourcesTreeMethods : LinksTreeMethodsBase
161 {
162     public LinksSourcesTreeMethods(ResizableDirectMemoryLinks<TLink> memory)
163         : base(memory)
164     {
165     }
166
167     protected override IntPtr GetLeftPointer(TLink node) =>
168 ↪ Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsSourceOffset;
169
170     protected override IntPtr GetRightPointer(TLink node) =>
171 ↪ Links.GetElement(LinkSizeInBytes, node) + Link.RightAsSourceOffset;
172
173     protected override TLink GetLeftValue(TLink node) =>
174 ↪ (Links.GetElement(LinkSizeInBytes, node) +
175 ↪ Link.LeftAsSourceOffset).GetValue<TLink>();
176
177     protected override TLink GetRightValue(TLink node) =>
178 ↪ (Links.GetElement(LinkSizeInBytes, node) +
179 ↪ Link.RightAsSourceOffset).GetValue<TLink>();
180
181     protected override TLink GetSize(TLink node)
182     {
183         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
184 ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
185         return Bit<TLink>.PartialRead(previousValue, 5, -5);
186     }
187
188     protected override void SetLeft(TLink node, TLink left) =>
189 ↪ (Links.GetElement(LinkSizeInBytes, node) +
190 ↪ Link.LeftAsSourceOffset).SetValue(left);

```

```

182     protected override void SetRight(TLink node, TLink right) =>
183         ↪ (Links.GetElement(LinkSizeInBytes, node) +
184         ↪ Link.RightAsSourceOffset).SetValue(right);
185
186     protected override void SetSize(TLink node, TLink size)
187     {
188         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
189         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
190         (Links.GetElement(LinkSizeInBytes, node) +
191         ↪ Link.SizeAsSourceOffset).SetValue(Bit<TLink>.PartialWrite(previousValue,
192         ↪ size, 5, -5));
193     }
194
195     protected override bool GetLeftIsChild(TLink node)
196     {
197         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
198         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
199         //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 4, 1);
200         return !_equalityComparer.Equals(Bit<TLink>.PartialRead(previousValue, 4, 1),
201         ↪ default);
202     }
203
204     protected override void SetLeftIsChild(TLink node, bool value)
205     {
206         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
207         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
208         var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 4,
209         ↪ 1);
210         (Links.GetElement(LinkSizeInBytes, node) +
211         ↪ Link.SizeAsSourceOffset).SetValue(modified);
212     }
213
214     protected override bool GetRightIsChild(TLink node)
215     {
216         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
217         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
218         //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 3, 1);
219         return !_equalityComparer.Equals(Bit<TLink>.PartialRead(previousValue, 3, 1),
220         ↪ default);
221     }
222
223     protected override void SetRightIsChild(TLink node, bool value)
224     {
225         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
226         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
227         var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 3,
228         ↪ 1);
229         (Links.GetElement(LinkSizeInBytes, node) +
230         ↪ Link.SizeAsSourceOffset).SetValue(modified);
231     }
232
233     protected override sbyte GetBalance(TLink node)
234     {
235         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
236         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
237         var value = (ulong)(Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 0, 3);
238         var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
239         ↪ 124 : value & 3);
240         return unpackedValue;
241     }
242
243     protected override void SetBalance(TLink node, sbyte value)
244     {
245         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
246         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
247         var packagedValue = (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) | value &
248         ↪ 3);
249         var modified = Bit<TLink>.PartialWrite(previousValue, packagedValue, 0, 3);
250         (Links.GetElement(LinkSizeInBytes, node) +
251         ↪ Link.SizeAsSourceOffset).SetValue(modified);
252     }
253
254     protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
255     {
256         var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
257         ↪ Link.SourceOffset).GetValue<TLink>();

```

```

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

```

```

289 }
290
291 private class LinksTargetsTreeMethods : LinksTreeMethodsBase
292 {
293     public LinksTargetsTreeMethods(ResizableDirectMemoryLinks<TLink> memory)
294         : base(memory)
295     {
296     }
297
298     protected override IntPtr GetLeftPointer(TLink node) =>
299         ↪ Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsTargetOffset;
300
301     protected override IntPtr GetRightPointer(TLink node) =>
302         ↪ Links.GetElement(LinkSizeInBytes, node) + Link.RightAsTargetOffset;
303
304     protected override TLink GetLeftValue(TLink node) =>
305         ↪ (Links.GetElement(LinkSizeInBytes, node) +
306         ↪ Link.LeftAsTargetOffset).GetValue<TLink>();
307
308     protected override TLink GetRightValue(TLink node) =>
309         ↪ (Links.GetElement(LinkSizeInBytes, node) +
310         ↪ Link.RightAsTargetOffset).GetValue<TLink>();
311
312     protected override TLink GetSize(TLink node)
313     {
314         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
315         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
316         return Bit<TLink>.PartialRead(previousValue, 5, -5);
317     }
318
319     protected override void SetLeft(TLink node, TLink left) =>
320         ↪ (Links.GetElement(LinkSizeInBytes, node) +
321         ↪ Link.LeftAsTargetOffset).SetValue(left);
322
323     protected override void SetRight(TLink node, TLink right) =>
324         ↪ (Links.GetElement(LinkSizeInBytes, node) +
325         ↪ Link.RightAsTargetOffset).SetValue(right);
326
327     protected override void SetSize(TLink node, TLink size)
328     {
329         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
330         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
331         (Links.GetElement(LinkSizeInBytes, node) +
332         ↪ Link.SizeAsTargetOffset).SetValue(Bit<TLink>.PartialWrite(previousValue,
333         ↪ size, 5, -5));
334     }
335
336     protected override bool GetLeftIsChild(TLink node)
337     {
338         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
339         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
340         //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 4, 1);
341         return !_equalityComparer.Equals(Bit<TLink>.PartialRead(previousValue, 4, 1),
342         ↪ default);
343     }
344
345     protected override void SetLeftIsChild(TLink node, bool value)
346     {
347         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
348         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
349         var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 4,
350         ↪ 1);
351         (Links.GetElement(LinkSizeInBytes, node) +
352         ↪ Link.SizeAsTargetOffset).SetValue(modified);
353     }
354
355     protected override bool GetRightIsChild(TLink node)
356     {
357         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
358         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
359         //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 3, 1);
360         return !_equalityComparer.Equals(Bit<TLink>.PartialRead(previousValue, 3, 1),
361         ↪ default);
362     }
363
364     protected override void SetRightIsChild(TLink node, bool value)
365     {

```



```

345     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
    ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
346     var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 3,
    ↪ 1);
347     (Links.GetElement(LinkSizeInBytes, node) +
    ↪ Link.SizeAsTargetOffset).SetValue(modified);
348 }
349
350 protected override sbyte GetBalance(TLink node)
351 {
352     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
    ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
353     var value = (ulong)(Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 0, 3);
354     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
    ↪ 124 : value & 3);
355     return unpackedValue;
356 }
357
358 protected override void SetBalance(TLink node, sbyte value)
359 {
360     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
    ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
361     var packagedValue = (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) | value &
    ↪ 3);
362     var modified = Bit<TLink>.PartialWrite(previousValue, packagedValue, 0, 3);
363     (Links.GetElement(LinkSizeInBytes, node) +
    ↪ Link.SizeAsTargetOffset).SetValue(modified);
364 }
365
366 protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
367 {
368     var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
    ↪ Link.TargetOffset).GetValue<TLink>();
369     var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
    ↪ Link.TargetOffset).GetValue<TLink>();
370     return LessThan(firstTarget, secondTarget) ||
371         (IsEquals(firstTarget, secondTarget) &&
    ↪ LessThan((Links.GetElement(LinkSizeInBytes, first) +
    ↪ Link.SourceOffset).GetValue<TLink>(),
    ↪ (Links.GetElement(LinkSizeInBytes, second) +
    ↪ Link.SourceOffset).GetValue<TLink>()));
372 }
373
374 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
375 {
376     var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
    ↪ Link.TargetOffset).GetValue<TLink>();
377     var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
    ↪ Link.TargetOffset).GetValue<TLink>();
378     return GreaterThan(firstTarget, secondTarget) ||
379         (IsEquals(firstTarget, secondTarget) &&
    ↪ GreaterThan((Links.GetElement(LinkSizeInBytes, first) +
    ↪ Link.SourceOffset).GetValue<TLink>(),
    ↪ (Links.GetElement(LinkSizeInBytes, second) +
    ↪ Link.SourceOffset).GetValue<TLink>()));
380 }
381
382 protected override TLink GetTreeRoot() => (Header +
    ↪ LinksHeader.FirstAsTargetOffset).GetValue<TLink>();
383
384 protected override TLink GetBasePartValue(TLink link) =>
    ↪ (Links.GetElement(LinkSizeInBytes, link) + Link.TargetOffset).GetValue<TLink>();
385 }
386 }
387 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Disposables;
5  using Platform.Collections.Arrays;
6  using Platform.Singletons;
7  using Platform.Memory;
8  using Platform.Data.Exceptions;
9  using Platform.Data.Constants;
10
11 #pragma warning disable 0649

```

```

12 #pragma warning disable 169
13 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
14
15 // ReSharper disable BuiltInTypeReferenceStyle
16
17 // #define ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
18
19 namespace Platform.Data.Doublets.ResizableDirectMemory
20 {
21     using id = UInt64;
22
23     public unsafe partial class UInt64ResizableDirectMemoryLinks : DisposableBase, ILinks<id>
24     {
25         /// <summary>Возвращает размер одной связи в байтах.</summary>
26         /// <remarks>
27         ///     Используется только во вне класса, не рекомендуется использовать внутри.
28         ///     Так как во вне не обязательно будет доступен unsafe C#.
29         /// </remarks>
30         public static readonly int LinkSizeInBytes = sizeof(Link);
31
32         public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
33
34         private struct Link
35         {
36             public id Source;
37             public id Target;
38             public id LeftAsSource;
39             public id RightAsSource;
40             public id SizeAsSource;
41             public id LeftAsTarget;
42             public id RightAsTarget;
43             public id SizeAsTarget;
44         }
45
46         private struct LinksHeader
47         {
48             public id AllocatedLinks;
49             public id ReservedLinks;
50             public id FreeLinks;
51             public id FirstFreeLink;
52             public id FirstAsSource;
53             public id FirstAsTarget;
54             public id LastFreeLink;
55             public id Reserved8;
56         }
57
58         private readonly long _memoryReservationStep;
59
60         private readonly IResizableDirectMemory _memory;
61         private LinksHeader* _header;
62         private Link* _links;
63
64         private LinksTargetsTreeMethods _targetsTreeMethods;
65         private LinksSourcesTreeMethods _sourcesTreeMethods;
66
67         // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
68         // → нужно использовать не список а дерево, так как так можно быстрее проверить на
69         // → наличие связи внутри
70         private UnusedLinksListMethods _unusedLinksListMethods;
71
72         /// <summary>
73         ///     Возвращает общее число связей находящихся в хранилище.
74         /// </summary>
75         private id Total => _header->AllocatedLinks - _header->FreeLinks;
76
77         // TODO: Дать возможность переопределять в конструкторе
78         public LinksCombinedConstants<id, id, int> Constants { get; }
79
80         public UInt64ResizableDirectMemoryLinks(string address) : this(address,
81             → DefaultLinksSizeStep) { }
82
83         /// <summary>
84         ///     Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
85         ///     → минимальным шагом расширения базы данных.
86         /// </summary>
87         /// <param name="address">Полный путь к файлу базы данных.</param>
88         /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
89         ///     → байтах.</param>
90         public UInt64ResizableDirectMemoryLinks(string address, long memoryReservationStep) :
91             → this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
92             → memoryReservationStep) { }
93     }
94 }

```

```

86 public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
87     ↳ DefaultLinksSizeStep) { }
88
89 public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
90     ↳ memoryReservationStep)
91 {
92     Constants = Default<LinksCombinedConstants<id, id, int>>.Instance;
93     _memory = memory;
94     _memoryReservationStep = memoryReservationStep;
95     if (memory.ReservedCapacity < memoryReservationStep)
96     {
97         memory.ReservedCapacity = memoryReservationStep;
98     }
99     SetPointers(_memory);
100     // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
101     _memory.UsedCapacity = ((long)_header->AllocatedLinks * sizeof(Link)) +
102     ↳ sizeof(LinksHeader);
103     // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
104     _header->ReservedLinks = (id)((_memory.ReservedCapacity - sizeof(LinksHeader)) /
105     ↳ sizeof(Link));
106 }
107
108 [MethodImpl(MethodImplOptions.AggressiveInlining)]
109 public id Count(IList<id> restrictions)
110 {
111     // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
112     if (restrictions.Count == 0)
113     {
114         return Total;
115     }
116     if (restrictions.Count == 1)
117     {
118         var index = restrictions[Constants.IndexPart];
119         if (index == Constants.Any)
120         {
121             return Total;
122         }
123         return Exists(index) ? 1UL : 0UL;
124     }
125     if (restrictions.Count == 2)
126     {
127         var index = restrictions[Constants.IndexPart];
128         var value = restrictions[1];
129         if (index == Constants.Any)
130         {
131             if (value == Constants.Any)
132             {
133                 return Total; // Any - как отсутствие ограничения
134             }
135             return _sourcesTreeMethods.CountUsages(value)
136                 + _targetsTreeMethods.CountUsages(value);
137         }
138         else
139         {
140             if (!Exists(index))
141             {
142                 return 0;
143             }
144             if (value == Constants.Any)
145             {
146                 return 1;
147             }
148             var storedLinkValue = GetLinkUnsafe(index);
149             if (storedLinkValue->Source == value ||
150                 storedLinkValue->Target == value)
151             {
152                 return 1;
153             }
154             return 0;
155         }
156     }
157     if (restrictions.Count == 3)
158     {
159         var index = restrictions[Constants.IndexPart];
160         var source = restrictions[Constants.SourcePart];
161         var target = restrictions[Constants.TargetPart];
162         if (index == Constants.Any)

```

```

160 {
161     if (source == Constants.Any && target == Constants.Any)
162     {
163         return Total;
164     }
165     else if (source == Constants.Any)
166     {
167         return _targetsTreeMethods.CountUsages(target);
168     }
169     else if (target == Constants.Any)
170     {
171         return _sourcesTreeMethods.CountUsages(source);
172     }
173     else //if(source != Any && target != Any)
174     {
175         // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
176         var link = _sourcesTreeMethods.Search(source, target);
177         return link == Constants.Null ? OUL : 1UL;
178     }
179 }
180 else
181 {
182     if (!Exists(index))
183     {
184         return 0;
185     }
186     if (source == Constants.Any && target == Constants.Any)
187     {
188         return 1;
189     }
190     var storedLinkValue = GetLinkUnsafe(index);
191     if (source != Constants.Any && target != Constants.Any)
192     {
193         if (storedLinkValue->Source == source &&
194             storedLinkValue->Target == target)
195         {
196             return 1;
197         }
198         return 0;
199     }
200     var value = default(id);
201     if (source == Constants.Any)
202     {
203         value = target;
204     }
205     if (target == Constants.Any)
206     {
207         value = source;
208     }
209     if (storedLinkValue->Source == value ||
210         storedLinkValue->Target == value)
211     {
212         return 1;
213     }
214     return 0;
215 }
216 }
217 throw new NotSupportedException("Другие размеры и способы ограничений не
↪ поддерживаются.");
218 }
219
220 [MethodImpl(MethodImplOptions.AggressiveInlining)]
221 public id Each(Func<IList<id>, id> handler, IList<id> restrictions)
222 {
223     if (restrictions.Count == 0)
224     {
225         for (id link = 1; link <= _header->AllocatedLinks; link++)
226         {
227             if (Exists(link))
228             {
229                 if (handler(GetLinkStruct(link)) == Constants.Break)
230                 {
231                     return Constants.Break;
232                 }
233             }
234         }
235         return Constants.Continue;
236     }
237     if (restrictions.Count == 1)

```

```

238 {
239     var index = restrictions[Constants.IndexPart];
240     if (index == Constants.Any)
241     {
242         return Each(handler, ArrayPool<ulong>.Empty);
243     }
244     if (!Exists(index))
245     {
246         return Constants.Continue;
247     }
248     return handler(GetLinkStruct(index));
249 }
250 if (restrictions.Count == 2)
251 {
252     var index = restrictions[Constants.IndexPart];
253     var value = restrictions[1];
254     if (index == Constants.Any)
255     {
256         if (value == Constants.Any)
257         {
258             return Each(handler, ArrayPool<ulong>.Empty);
259         }
260         if (Each(handler, new[] { index, value, Constants.Any }) == Constants.Break)
261         {
262             return Constants.Break;
263         }
264         return Each(handler, new[] { index, Constants.Any, value });
265     }
266     else
267     {
268         if (!Exists(index))
269         {
270             return Constants.Continue;
271         }
272         if (value == Constants.Any)
273         {
274             return handler(GetLinkStruct(index));
275         }
276         var storedLinkValue = GetLinkUnsafe(index);
277         if (storedLinkValue->Source == value ||
278             storedLinkValue->Target == value)
279         {
280             return handler(GetLinkStruct(index));
281         }
282         return Constants.Continue;
283     }
284 }
285 if (restrictions.Count == 3)
286 {
287     var index = restrictions[Constants.IndexPart];
288     var source = restrictions[Constants.SourcePart];
289     var target = restrictions[Constants.TargetPart];
290     if (index == Constants.Any)
291     {
292         if (source == Constants.Any && target == Constants.Any)
293         {
294             return Each(handler, ArrayPool<ulong>.Empty);
295         }
296         else if (source == Constants.Any)
297         {
298             return _targetsTreeMethods.EachReference(target, handler);
299         }
300         else if (target == Constants.Any)
301         {
302             return _sourcesTreeMethods.EachReference(source, handler);
303         }
304         else //if(source != Any && target != Any)
305         {
306             var link = _sourcesTreeMethods.Search(source, target);
307             return link == Constants.Null ? Constants.Continue :
308                 ↪ handler(GetLinkStruct(link));
309         }
310     }
311     else
312     {
313         if (!Exists(index))
314         {
315             return Constants.Continue;
316         }

```

```

315     }
316     if (source == Constants.Any && target == Constants.Any)
317     {
318         return handler(GetLinkStruct(index));
319     }
320     var storedLinkValue = GetLinkUnsafe(index);
321     if (source != Constants.Any && target != Constants.Any)
322     {
323         if (storedLinkValue->Source == source &&
324             storedLinkValue->Target == target)
325         {
326             return handler(GetLinkStruct(index));
327         }
328         return Constants.Continue;
329     }
330     var value = default(id);
331     if (source == Constants.Any)
332     {
333         value = target;
334     }
335     if (target == Constants.Any)
336     {
337         value = source;
338     }
339     if (storedLinkValue->Source == value ||
340         storedLinkValue->Target == value)
341     {
342         return handler(GetLinkStruct(index));
343     }
344     return Constants.Continue;
345 }
346 }
347 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↳ поддерживаются.");
348 }
349
350 /// <remarks>
351 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
    ↳ в другом месте (но не в менеджере памяти, а в логике Links)
352 /// </remarks>
353 [MethodImpl(MethodImplOptions.AggressiveInlining)]
354 public id Update(ICollection<id> values)
355 {
356     var linkIndex = values[Constants.IndexPart];
357     var link = GetLinkUnsafe(linkIndex);
358     // Будет корректно работать только в том случае, если пространство выделенной связи
    ↳ предварительно заполнено нулями
359     if (link->Source != Constants.Null)
360     {
361         _sourcesTreeMethods.Detach(new IntPtr(&_header->FirstAsSource), linkIndex);
362     }
363     if (link->Target != Constants.Null)
364     {
365         _targetsTreeMethods.Detach(new IntPtr(&_header->FirstAsTarget), linkIndex);
366     }
367 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
368     var leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
369     var rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
370     if (leftTreeSize != rightTreeSize)
371     {
372         throw new Exception("One of the trees is broken.");
373     }
374 #endif
375     link->Source = values[Constants.SourcePart];
376     link->Target = values[Constants.TargetPart];
377     if (link->Source != Constants.Null)
378     {
379         _sourcesTreeMethods.Attach(new IntPtr(&_header->FirstAsSource), linkIndex);
380     }
381     if (link->Target != Constants.Null)
382     {
383         _targetsTreeMethods.Attach(new IntPtr(&_header->FirstAsTarget), linkIndex);
384     }
385 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
386     leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
387     rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
388     if (leftTreeSize != rightTreeSize)
389     {

```

```

390         throw new Exception("One of the trees is broken.");
391     }
392 #endif
393     return linkIndex;
394 }
395
396 [MethodImpl(MethodImplOptions.AggressiveInlining)]
397 private IList<id> GetLinkStruct(id linkIndex)
398 {
399     var link = GetLinkUnsafe(linkIndex);
400     return new UInt64Link(linkIndex, link->Source, link->Target);
401 }
402
403 [MethodImpl(MethodImplOptions.AggressiveInlining)]
404 private Link* GetLinkUnsafe(id linkIndex) => &_amp;_links[linkIndex];
405
406 /// <remarks>
407 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
408   ↳ пространство
409 /// </remarks>
410 public id Create()
411 {
412     var freeLink = _header->FirstFreeLink;
413     if (freeLink != Constants.Null)
414     {
415         _unusedLinksListMethods.Detach(freeLink);
416     }
417     else
418     {
419         if (_header->AllocatedLinks > Constants.MaxPossibleIndex)
420         {
421             throw new LinksLimitReachedException(Constants.MaxPossibleIndex);
422         }
423         if (_header->AllocatedLinks >= _header->ReservedLinks - 1)
424         {
425             _memory.ReservedCapacity += _memory.ReservationStep;
426             SetPointers(_memory);
427             _header->ReservedLinks = (id)(_memory.ReservedCapacity / sizeof(Link));
428         }
429         _header->AllocatedLinks++;
430         _memory.UsedCapacity += sizeof(Link);
431         freeLink = _header->AllocatedLinks;
432     }
433     return freeLink;
434 }
435
436 public void Delete(id link)
437 {
438     if (link < _header->AllocatedLinks)
439     {
440         _unusedLinksListMethods.AttachAsFirst(link);
441     }
442     else if (link == _header->AllocatedLinks)
443     {
444         _header->AllocatedLinks--;
445         _memory.UsedCapacity -= sizeof(Link);
446         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
447         // ↳ пока не дойдём до первой существующей связи
448         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
449         while (_header->AllocatedLinks > 0 && IsUnusedLink(_header->AllocatedLinks))
450         {
451             _unusedLinksListMethods.Detach(_header->AllocatedLinks);
452             _header->AllocatedLinks--;
453             _memory.UsedCapacity -= sizeof(Link);
454         }
455     }
456 }
457
458 /// <remarks>
459 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
460   ↳ адрес реально поменялся
461 ///
462 /// Указатель this.links может быть в том же месте,
463 /// так как 0-я связь не используется и имеет такой же размер как Header,
464 /// поэтому header размещается в том же месте, что и 0-я связь
465 /// </remarks>
466 private void SetPointers(IResizableDirectMemory memory)
467 {
468     if (memory == null)

```

```

466     {
467         _header = null;
468         _links = null;
469         _unusedLinksListMethods = null;
470         _targetsTreeMethods = null;
471         _unusedLinksListMethods = null;
472     }
473     else
474     {
475         _header = (LinksHeader*)(void*)memory.Pointer;
476         _links = (Link*)(void*)memory.Pointer;
477         _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
478         _targetsTreeMethods = new LinksTargetsTreeMethods(this);
479         _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
480     }
481 }
482
483 [MethodImpl(MethodImplOptions.AggressiveInlining)]
484 private bool Exists(id link) => link >= Constants.MinPossibleIndex && link <=
    ↳ _header->AllocatedLinks && !IsUnusedLink(link);
485
486 [MethodImpl(MethodImplOptions.AggressiveInlining)]
487 private bool IsUnusedLink(id link) => _header->FirstFreeLink == link
488     || (_links[link].SizeAsSource == Constants.Null &&
    ↳ _links[link].Source != Constants.Null);
489
490 #region Disposable
491
492 protected override bool AllowMultipleDisposeCalls => true;
493
494 protected override void Dispose(bool manual, bool wasDisposed)
495 {
496     if (!wasDisposed)
497     {
498         SetPointers(null);
499         _memory.DisposeIfPossible();
500     }
501 }
502
503 #endregion
504 }
505 }

```

```

./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.ListMethods.cs
1  using Platform.Collections.Methods.Lists;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.ResizableDirectMemory
6  {
7      unsafe partial class UInt64ResizableDirectMemoryLinks
8      {
9          private class UnusedLinksListMethods : CircularDoublyLinkedListMethods<ulong>
10         {
11             private readonly Link* _links;
12             private readonly LinksHeader* _header;
13
14             public UnusedLinksListMethods(Link* links, LinksHeader* header)
15             {
16                 _links = links;
17                 _header = header;
18             }
19
20             protected override ulong GetFirst() => _header->FirstFreeLink;
21
22             protected override ulong GetLast() => _header->LastFreeLink;
23
24             protected override ulong GetPrevious(ulong element) => _links[element].Source;
25
26             protected override ulong GetNext(ulong element) => _links[element].Target;
27
28             protected override ulong GetSize() => _header->FreeLinks;
29
30             protected override void SetFirst(ulong element) => _header->FirstFreeLink = element;
31
32             protected override void SetLast(ulong element) => _header->LastFreeLink = element;
33
34             protected override void SetPrevious(ulong element, ulong previous) =>
    ↳ _links[element].Source = previous;
35

```



```

36         protected override void SetNext(ulong element, ulong next) => _links[element].Target
           ↪ = next;
37
38         protected override void SetSize(ulong size) => _header->FreeLinks = size;
39     }
40 }
41 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using System.Text;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Data.Constants;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.ResizableDirectMemory
11 {
12     unsafe partial class UInt64ResizableDirectMemoryLinks
13     {
14         private abstract class LinksTreeMethodsBase :
           ↪ SizedAndThreadedAVLBalancedTreeMethods<ulong>
15         {
16             private readonly UInt64ResizableDirectMemoryLinks _memory;
17             private readonly LinksCombinedConstants<ulong, ulong, int> _constants;
18             protected readonly Link* Links;
19             protected readonly LinksHeader* Header;
20
21             protected LinksTreeMethodsBase(UInt64ResizableDirectMemoryLinks memory)
22             {
23                 Links = memory._links;
24                 Header = memory._header;
25                 _memory = memory;
26                 _constants = memory.Constants;
27             }
28
29             [MethodImpl(MethodImplOptions.AggressiveInlining)]
30             protected abstract ulong GetTreeRoot();
31
32             [MethodImpl(MethodImplOptions.AggressiveInlining)]
33             protected abstract ulong GetBasePartValue(ulong link);
34
35             public ulong this[ulong index]
36             {
37                 get
38                 {
39                     var root = GetTreeRoot();
40                     if (index >= GetSize(root))
41                     {
42                         return 0;
43                     }
44                     while (root != 0)
45                     {
46                         var left = GetLeftOrDefault(root);
47                         var leftSize = GetSizeOrZero(left);
48                         if (index < leftSize)
49                         {
50                             root = left;
51                             continue;
52                         }
53                         if (index == leftSize)
54                         {
55                             return root;
56                         }
57                         root = GetRightOrDefault(root);
58                         index -= leftSize + 1;
59                     }
60                     return 0; // TODO: Impossible situation exception (only if tree structure
           ↪ broken)
61                 }
62             }
63
64             // TODO: Return indices range instead of references count
65             public ulong CountUsages(ulong link)
66             {
67                 var root = GetTreeRoot();
68                 var total = GetSize(root);
69                 var totalRightIgnore = 0UL;

```

```

70     while (root != 0)
71     {
72         var @base = GetBasePartValue(root);
73         if (@base <= link)
74         {
75             root = GetRightOrDefault(root);
76         }
77         else
78         {
79             totalRightIgnore += GetRightSize(root) + 1;
80             root = GetLeftOrDefault(root);
81         }
82     }
83     root = GetTreeRoot();
84     var totalLeftIgnore = 0UL;
85     while (root != 0)
86     {
87         var @base = GetBasePartValue(root);
88         if (@base >= link)
89         {
90             root = GetLeftOrDefault(root);
91         }
92         else
93         {
94             totalLeftIgnore += GetLeftSize(root) + 1;
95             root = GetRightOrDefault(root);
96         }
97     }
98     return total - totalRightIgnore - totalLeftIgnore;
99 }
100
101 public ulong EachReference(ulong link, Func<IList<ulong>, ulong> handler)
102 {
103     var root = GetTreeRoot();
104     if (root == 0)
105     {
106         return _constants.Continue;
107     }
108     ulong first = 0, current = root;
109     while (current != 0)
110     {
111         var @base = GetBasePartValue(current);
112         if (@base >= link)
113         {
114             if (@base == link)
115             {
116                 first = current;
117             }
118             current = GetLeftOrDefault(current);
119         }
120         else
121         {
122             current = GetRightOrDefault(current);
123         }
124     }
125     if (first != 0)
126     {
127         current = first;
128         while (true)
129         {
130             if (handler(_memory.GetLinkStruct(current)) == _constants.Break)
131             {
132                 return _constants.Break;
133             }
134             current = GetNext(current);
135             if (current == 0 || GetBasePartValue(current) != link)
136             {
137                 break;
138             }
139         }
140     }
141     return _constants.Continue;
142 }
143
144 protected override void PrintNodeValue(ulong node, StringBuilder sb)
145 {
146     sb.Append(' ');
147     sb.Append(Links[node].Source);
148     sb.Append('-');

```

```

149         sb.Append('>');
150         sb.Append(Links[node].Target);
151     }
152 }
153
154 private class LinksSourcesTreeMethods : LinksTreeMethodsBase
155 {
156     public LinksSourcesTreeMethods(UInt64ResizableDirectMemoryLinks memory)
157         : base(memory)
158     {
159     }
160
161     protected override IntPtr GetLeftPointer(ulong node) => new
162         ↳ IntPtr(&Links[node].LeftAsSource);
163
164     protected override IntPtr GetRightPointer(ulong node) => new
165         ↳ IntPtr(&Links[node].RightAsSource);
166
167     protected override ulong GetLeftValue(ulong node) => Links[node].LeftAsSource;
168
169     protected override ulong GetRightValue(ulong node) => Links[node].RightAsSource;
170
171     protected override ulong GetSize(ulong node)
172     {
173         var previousValue = Links[node].SizeAsSource;
174         //return Math.PartialRead(previousValue, 5, -5);
175         return (previousValue & 4294967264) >> 5;
176     }
177
178     protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource
179         ↳ = left;
180
181     protected override void SetRight(ulong node, ulong right) =>
182         ↳ Links[node].RightAsSource = right;
183
184     protected override void SetSize(ulong node, ulong size)
185     {
186         var previousValue = Links[node].SizeAsSource;
187         //var modified = Math.PartialWrite(previousValue, size, 5, -5);
188         var modified = (previousValue & 31) | ((size & 134217727) << 5);
189         Links[node].SizeAsSource = modified;
190     }
191
192     protected override bool GetLeftIsChild(ulong node)
193     {
194         var previousValue = Links[node].SizeAsSource;
195         //return (Integer)Math.PartialRead(previousValue, 4, 1);
196         return (previousValue & 16) >> 4 == 1UL;
197     }
198
199     protected override void SetLeftIsChild(ulong node, bool value)
200     {
201         var previousValue = Links[node].SizeAsSource;
202         //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 4, 1);
203         var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
204         Links[node].SizeAsSource = modified;
205     }
206
207     protected override bool GetRightIsChild(ulong node)
208     {
209         var previousValue = Links[node].SizeAsSource;
210         //return (Integer)Math.PartialRead(previousValue, 3, 1);
211         return (previousValue & 8) >> 3 == 1UL;
212     }
213
214     protected override void SetRightIsChild(ulong node, bool value)
215     {
216         var previousValue = Links[node].SizeAsSource;
217         //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 3, 1);
218         var modified = (previousValue & 4294967287) | ((value ? 1UL : 0UL) << 3);
219         Links[node].SizeAsSource = modified;
220     }
221
222     protected override sbyte GetBalance(ulong node)
223     {
224         var previousValue = Links[node].SizeAsSource;
225         //var value = Math.PartialRead(previousValue, 0, 3);
226         var value = previousValue & 7;

```

```

223     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
224         ↪ 124 : value & 3);
225     return unpackedValue;
226 }
227
228 protected override void SetBalance(ulong node, sbyte value)
229 {
230     var previousValue = Links[node].SizeAsSource;
231     var packagedValue = (ulong)((((byte)value >> 5) & 4) | value & 3);
232     //var modified = Math.PartialWrite(previousValue, packagedValue, 0, 3);
233     var modified = (previousValue & 4294967288) | (packagedValue & 7);
234     Links[node].SizeAsSource = modified;
235 }
236
237 protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
238     => Links[first].Source < Links[second].Source ||
239     (Links[first].Source == Links[second].Source && Links[first].Target <
240     ↪ Links[second].Target);
241
242 protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
243     => Links[first].Source > Links[second].Source ||
244     (Links[first].Source == Links[second].Source && Links[first].Target >
245     ↪ Links[second].Target);
246
247 protected override ulong GetTreeRoot() => Header->FirstAsSource;
248
249 protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
250
251 /// <summary>
252 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
253 ↪ (концом)
254 /// по дереву (индексу) связей, отсортированному по Source, а затем по Target.
255 /// </summary>
256 /// <param name="source">Индекс связи, которая является началом на искомой
257 ↪ связи.</param>
258 /// <param name="target">Индекс связи, которая является концом на искомой
259 ↪ связи.</param>
260 /// <returns>Индекс искомой связи.</returns>
261 public ulong Search(ulong source, ulong target)
262 {
263     var root = Header->FirstAsSource;
264     while (root != 0)
265     {
266         var rootSource = Links[root].Source;
267         var rootTarget = Links[root].Target;
268         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
269             ↪ node.Key < root.Key
270         {
271             root = GetLeftOrDefault(root);
272         }
273         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget))
274             ↪ // node.Key > root.Key
275         {
276             root = GetRightOrDefault(root);
277         }
278         else // node.Key == root.Key
279         {
280             return root;
281         }
282     }
283     return 0;
284 }
285
286 [MethodImpl(MethodImplOptions.AggressiveInlining)]
287 private static bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
288     ↪ ulong secondSource, ulong secondTarget)
289     => firstSource < secondSource || (firstSource == secondSource && firstTarget <
290     ↪ secondTarget);
291
292 [MethodImpl(MethodImplOptions.AggressiveInlining)]
293 private static bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
294     ↪ ulong secondSource, ulong secondTarget)
295     => firstSource > secondSource || (firstSource == secondSource && firstTarget >
296     ↪ secondTarget);
297
298 [MethodImpl(MethodImplOptions.AggressiveInlining)]
299 protected override void ClearNode(ulong node)
300 {

```

```

289         Links[node].LeftAsSource = OUL;
290         Links[node].RightAsSource = OUL;
291         Links[node].SizeAsSource = OUL;
292     }
293
294     [MethodImpl(MethodImplOptions.AggressiveInlining)]
295     protected override ulong GetZero() => OUL;
296
297     [MethodImpl(MethodImplOptions.AggressiveInlining)]
298     protected override ulong GetOne() => 1UL;
299
300     [MethodImpl(MethodImplOptions.AggressiveInlining)]
301     protected override ulong GetTwo() => 2UL;
302
303     [MethodImpl(MethodImplOptions.AggressiveInlining)]
304     protected override bool ValueEqualToZero(IntPtr pointer) =>
305         ↳ *(ulong*)pointer.ToPointer() == OUL;
306
307     [MethodImpl(MethodImplOptions.AggressiveInlining)]
308     protected override bool EqualToZero(ulong value) => value == OUL;
309
310     [MethodImpl(MethodImplOptions.AggressiveInlining)]
311     protected override bool IsEquals(ulong first, ulong second) => first == second;
312
313     [MethodImpl(MethodImplOptions.AggressiveInlining)]
314     protected override bool GreaterThanZero(ulong value) => value > OUL;
315
316     [MethodImpl(MethodImplOptions.AggressiveInlining)]
317     protected override bool GreaterThan(ulong first, ulong second) => first > second;
318
319     [MethodImpl(MethodImplOptions.AggressiveInlining)]
320     protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >=
321         ↳ second;
322
323     [MethodImpl(MethodImplOptions.AggressiveInlining)]
324     protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0
325         ↳ is always true for ulong
326
327     [MethodImpl(MethodImplOptions.AggressiveInlining)]
328     protected override bool LessOrEqualThanZero(ulong value) => value == 0; // value is
329         ↳ always >= 0 for ulong
330
331     [MethodImpl(MethodImplOptions.AggressiveInlining)]
332     protected override bool LessOrEqualThan(ulong first, ulong second) => first <=
333         ↳ second;
334
335     [MethodImpl(MethodImplOptions.AggressiveInlining)]
336     protected override bool LessThanZero(ulong value) => false; // value < 0 is always
337         ↳ false for ulong
338
339     [MethodImpl(MethodImplOptions.AggressiveInlining)]
340     protected override bool LessThan(ulong first, ulong second) => first < second;
341
342     [MethodImpl(MethodImplOptions.AggressiveInlining)]
343     protected override ulong Increment(ulong value) => ++value;
344
345     [MethodImpl(MethodImplOptions.AggressiveInlining)]
346     protected override ulong Decrement(ulong value) => --value;
347
348     [MethodImpl(MethodImplOptions.AggressiveInlining)]
349     protected override ulong Add(ulong first, ulong second) => first + second;
350
351     [MethodImpl(MethodImplOptions.AggressiveInlining)]
352     protected override ulong Subtract(ulong first, ulong second) => first - second;
353 }
354
355 private class LinksTargetsTreeMethods : LinksTreeMethodsBase
356 {
357     public LinksTargetsTreeMethods(UInt64ResizableDirectMemoryLinks memory)
358         : base(memory)
359     {
360     }
361
362     //protected override IntPtr GetLeft(ulong node) => new
363     ↳ IntPtr(&Links[node].LeftAsTarget);
364
365     //protected override IntPtr GetRight(ulong node) => new
366     ↳ IntPtr(&Links[node].RightAsTarget);
367 }

```

```

360 //protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;
361
362 //protected override void SetLeft(ulong node, ulong left) =>
363     ↳ Links[node].LeftAsTarget = left;
364
365 //protected override void SetRight(ulong node, ulong right) =>
366     ↳ Links[node].RightAsTarget = right;
367
368 //protected override void SetSize(ulong node, ulong size) =>
369     ↳ Links[node].SizeAsTarget = size;
370
371 protected override IntPtr GetLeftPointer(ulong node) => new
372     ↳ IntPtr(&Links[node].LeftAsTarget);
373
374 protected override IntPtr GetRightPointer(ulong node) => new
375     ↳ IntPtr(&Links[node].RightAsTarget);
376
377 protected override ulong GetLeftValue(ulong node) => Links[node].LeftAsTarget;
378
379 protected override ulong GetRightValue(ulong node) => Links[node].RightAsTarget;
380
381 protected override ulong GetSize(ulong node)
382 {
383     var previousValue = Links[node].SizeAsTarget;
384     //return Math.PartialRead(previousValue, 5, -5);
385     return (previousValue & 4294967264) >> 5;
386 }
387
388 protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget
389     ↳ = left;
390
391 protected override void SetRight(ulong node, ulong right) =>
392     ↳ Links[node].RightAsTarget = right;
393
394 protected override void SetSize(ulong node, ulong size)
395 {
396     var previousValue = Links[node].SizeAsTarget;
397     //var modified = Math.PartialWrite(previousValue, size, 5, -5);
398     var modified = (previousValue & 31) | ((size & 134217727) << 5);
399     Links[node].SizeAsTarget = modified;
400 }
401
402 protected override bool GetLeftIsChild(ulong node)
403 {
404     var previousValue = Links[node].SizeAsTarget;
405     //return (Integer)Math.PartialRead(previousValue, 4, 1);
406     return (previousValue & 16) >> 4 == 1UL;
407     // TODO: Check if this is possible to use
408     //var nodeSize = GetSize(node);
409     //var left = GetLeftValue(node);
410     //var leftSize = GetSizeOrZero(left);
411     //return leftSize > 0 && nodeSize > leftSize;
412 }
413
414 protected override void SetLeftIsChild(ulong node, bool value)
415 {
416     var previousValue = Links[node].SizeAsTarget;
417     //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 4, 1);
418     var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
419     Links[node].SizeAsTarget = modified;
420 }
421
422 protected override bool GetRightIsChild(ulong node)
423 {
424     var previousValue = Links[node].SizeAsTarget;
425     //return (Integer)Math.PartialRead(previousValue, 3, 1);
426     return (previousValue & 8) >> 3 == 1UL;
427     // TODO: Check if this is possible to use
428     //var nodeSize = GetSize(node);
429     //var right = GetRightValue(node);
430     //var rightSize = GetSizeOrZero(right);
431     //return rightSize > 0 && nodeSize > rightSize;
432 }
433
434 protected override void SetRightIsChild(ulong node, bool value)
435 {
436     var previousValue = Links[node].SizeAsTarget;
437     //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 3, 1);

```

```

431     var modified = (previousValue & 4294967287) | (((value ? 1UL : 0UL) << 3);
432     Links[node].SizeAsTarget = modified;
433 }
434
435 protected override sbyte GetBalance(ulong node)
436 {
437     var previousValue = Links[node].SizeAsTarget;
438     //var value = Math.PartialRead(previousValue, 0, 3);
439     var value = previousValue & 7;
440     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
441     ↪ 124 : value & 3);
442     return unpackedValue;
443 }
444
445 protected override void SetBalance(ulong node, sbyte value)
446 {
447     var previousValue = Links[node].SizeAsTarget;
448     var packagedValue = (ulong)((((byte)value >> 5) & 4) | value & 3);
449     //var modified = Math.PartialWrite(previousValue, packagedValue, 0, 3);
450     var modified = (previousValue & 4294967288) | (packagedValue & 7);
451     Links[node].SizeAsTarget = modified;
452 }
453
454 protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
455     => Links[first].Target < Links[second].Target ||
456     (Links[first].Target == Links[second].Target && Links[first].Source <
457     ↪ Links[second].Source);
458
459 protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
460     => Links[first].Target > Links[second].Target ||
461     (Links[first].Target == Links[second].Target && Links[first].Source >
462     ↪ Links[second].Source);
463
464 protected override ulong GetTreeRoot() => Header->FirstAsTarget;
465
466 protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
467
468 [MethodImpl(MethodImplOptions.AggressiveInlining)]
469 protected override void ClearNode(ulong node)
470 {
471     Links[node].LeftAsTarget = 0UL;
472     Links[node].RightAsTarget = 0UL;
473     Links[node].SizeAsTarget = 0UL;
474 }
475 }
476 }
477 }

```

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

```

1 using System.Collections.Generic;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Sequences.Converters
6 {
7     public class BalancedVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
8     {
9         public BalancedVariantConverter(ILinks<TLink> links) : base(links) { }
10
11         public override TLink Convert(ICollection<TLink> sequence)
12         {
13             var length = sequence.Count;
14             if (length < 1)
15             {
16                 return default;
17             }
18             if (length == 1)
19             {
20                 return sequence[0];
21             }
22             // Make copy of next layer
23             if (length > 2)
24             {
25                 // TODO: Try to use stackalloc (which at the moment is not working with
26                 ↪ generics) but will be possible with Sigil
27                 var halvedSequence = new TLink[(length / 2) + (length % 2)];
28                 HalveSequence(halvedSequence, sequence, length);
29                 sequence = halvedSequence;
30                 length = halvedSequence.Length;
31             }
32         }
33     }
34 }

```

```

30     }
31     // Keep creating layer after layer
32     while (length > 2)
33     {
34         HalveSequence(sequence, sequence, length);
35         length = (length / 2) + (length % 2);
36     }
37     return Links.GetOrCreate(sequence[0], sequence[1]);
38 }
39
40 private void HalveSequence(IList<TLink> destination, IList<TLink> source, int length)
41 {
42     var loopedLength = length - (length % 2);
43     for (var i = 0; i < loopedLength; i += 2)
44     {
45         destination[i / 2] = Links.GetOrCreate(source[i], source[i + 1]);
46     }
47     if (length > loopedLength)
48     {
49         destination[length / 2] = source[length - 1];
50     }
51 }
52 }
53 }

```

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

```

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

```



```

49     {
50     }
51
52     public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
    ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
    ↪ doInitialFrequenciesIncrement)
53         : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One,
    ↪ doInitialFrequenciesIncrement)
54     {
55     }
56
57     public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
    ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
    ↪ minFrequencyToCompress, bool doInitialFrequenciesIncrement)
58         : base(links)
59     {
60         _baseConverter = baseConverter;
61         _doubletFrequenciesCache = doubletFrequenciesCache;
62         if (_comparer.Compare(minFrequencyToCompress, Integer<TLink>.One) < 0)
63         {
64             minFrequencyToCompress = Integer<TLink>.One;
65         }
66         _minFrequencyToCompress = minFrequencyToCompress;
67         _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
68         ResetMaxDoublet();
69     }
70
71     public override TLink Convert(IList<TLink> source) =>
    ↪ _baseConverter.Convert(Compress(source));
72
73     /// <remarks>
74     /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding .
75     /// Faster version (doublets' frequencies dictionary is not recreated).
76     /// </remarks>
77     private IList<TLink> Compress(IList<TLink> sequence)
78     {
79         if (sequence.IsNullOrEmpty())
80         {
81             return null;
82         }
83         if (sequence.Count == 1)
84         {
85             return sequence;
86         }
87         if (sequence.Count == 2)
88         {
89             return new[] { Links.GetOrCreate(sequence[0], sequence[1]) };
90         }
91         // TODO: arraypool with min size (to improve cache locality) or stackalloc with Sigil
92         var copy = new HalfDoublet[sequence.Count];
93         Doublet<TLink> doublet = default;
94         for (var i = 1; i < sequence.Count; i++)
95         {
96             doublet.Source = sequence[i - 1];
97             doublet.Target = sequence[i];
98             LinkFrequency<TLink> data;
99             if (_doInitialFrequenciesIncrement)
100             {
101                 data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
102             }
103             else
104             {
105                 data = _doubletFrequenciesCache.GetFrequency(ref doublet);
106                 if (data == null)
107                 {
108                     throw new NotSupportedException("If you ask not to increment
    ↪ frequencies, it is expected that all frequencies for the sequence
    ↪ are prepared.");
109                 }
110             }
111             copy[i - 1].Element = sequence[i - 1];
112             copy[i - 1].DoubletData = data;
113             UpdateMaxDoublet(ref doublet, data);
114         }
115         copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
116         copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
117         if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
118         {

```

```

119     var newLength = ReplaceDoublets(copy);
120     sequence = new TLink[newLength];
121     for (int i = 0; i < newLength; i++)
122     {
123         sequence[i] = copy[i].Element;
124     }
125 }
126 return sequence;
127 }
128
129 /// <remarks>
130 /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding
131 /// </remarks>
132 private int ReplaceDoublets(HalfDoublet[] copy)
133 {
134     var oldLength = copy.Length;
135     var newLength = copy.Length;
136     while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
137     {
138         var maxDoubletSource = _maxDoublet.Source;
139         var maxDoubletTarget = _maxDoublet.Target;
140         if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
141         {
142             _maxDoubletData.Link = Links.GetOrCreate(maxDoubletSource, maxDoubletTarget);
143         }
144         var maxDoubletReplacementLink = _maxDoubletData.Link;
145         oldLength--;
146         var oldLengthMinusTwo = oldLength - 1;
147         // Substitute all usages
148         int w = 0, r = 0; // (r == read, w == write)
149         for (; r < oldLength; r++)
150         {
151             if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
152                 ↪ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
153             {
154                 if (r > 0)
155                 {
156                     var previous = copy[w - 1].Element;
157                     copy[w - 1].DoubletData.DecrementFrequency();
158                     copy[w - 1].DoubletData =
159                         ↪ _doubletFrequenciesCache.IncrementFrequency(previous,
160                             ↪ maxDoubletReplacementLink);
161                 }
162                 if (r < oldLengthMinusTwo)
163                 {
164                     var next = copy[r + 2].Element;
165                     copy[r + 1].DoubletData.DecrementFrequency();
166                     copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(maxDoubletReplacementLink,
167                         ↪ next);
168                 }
169                 copy[w++] = maxDoubletReplacementLink;
170                 r++;
171                 newLength--;
172             }
173             else
174             {
175                 copy[w++] = copy[r];
176             }
177         }
178         if (w < newLength)
179         {
180             copy[w] = copy[r];
181         }
182         oldLength = newLength;
183         ResetMaxDoublet();
184         UpdateMaxDoublet(copy, newLength);
185     }
186     return newLength;
187 }
188
189 [MethodImpl(MethodImplOptions.AggressiveInlining)]
190 private void ResetMaxDoublet()
191 {
192     _maxDoublet = new Doublet<TLink>();
193     _maxDoubletData = new LinkFrequency<TLink>();
194 }
195
196 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

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

```

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

```

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

```

1 using System.Collections.Generic;
2 using System.Linq;
3 using Platform.Interfaces;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Sequences.Converters
8 {
9     public class OptimalVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12         ↪ EqualityComparer<TLink>.Default;
13         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
14
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             {

```

```

24         return sequence[0];
25     }
26     var links = Links;
27     if (length == 2)
28     {
29         return links.GetOrCreate(sequence[0], sequence[1]);
30     }
31     sequence = sequence.ToArray();
32     var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
33     while (length > 2)
34     {
35         var levelRepeat = 1;
36         var currentLevel = levels[0];
37         var previousLevel = levels[0];
38         var skipOnce = false;
39         var w = 0;
40         for (var i = 1; i < length; i++)
41         {
42             if (_equalityComparer.Equals(currentLevel, levels[i]))
43             {
44                 levelRepeat++;
45                 skipOnce = false;
46                 if (levelRepeat == 2)
47                 {
48                     sequence[w] = links.GetOrCreate(sequence[i - 1], sequence[i]);
49                     var newLevel = i >= length - 1 ?
50                         GetPreviousLowerThanCurrentOrCurrent(previousLevel,
51                             ↪ currentLevel) :
52                         i < 2 ?
53                         GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
54                         GetGreatestNeighbourLowerThanCurrentOrCurrent(previousLevel,
55                             ↪ currentLevel, levels[i + 1]);
56                     levels[w] = newLevel;
57                     previousLevel = currentLevel;
58                     w++;
59                     levelRepeat = 0;
60                     skipOnce = true;
61                 }
62                 else if (i == length - 1)
63                 {
64                     sequence[w] = sequence[i];
65                     levels[w] = levels[i];
66                     w++;
67                 }
68             }
69             else
70             {
71                 currentLevel = levels[i];
72                 levelRepeat = 1;
73                 if (skipOnce)
74                 {
75                     skipOnce = false;
76                 }
77                 else
78                 {
79                     sequence[w] = sequence[i - 1];
80                     levels[w] = levels[i - 1];
81                     previousLevel = levels[w];
82                     w++;
83                 }
84                 if (i == length - 1)
85                 {
86                     sequence[w] = sequence[i];
87                     levels[w] = levels[i];
88                     w++;
89                 }
90             }
91         }
92         length = w;
93     }
94     return links.GetOrCreate(sequence[0], sequence[1]);
95 }
96
97 private static TLink GetGreatestNeighbourLowerThanCurrentOrCurrent(TLink previous, TLink
98     ↪ current, TLink next)
99 {
100     return _comparer.Compare(previous, next) > 0
101         ? _comparer.Compare(previous, current) < 0 ? previous : current
102         : _comparer.Compare(next, current) < 0 ? next : current;

```

```

100     }
101
102     private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
        ↪ _comparer.Compare(next, current) < 0 ? next : current;
103
104     private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
        ↪ => _comparer.Compare(previous, current) < 0 ? previous : current;
105 }
106 }

```

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

```

1 using System.Collections.Generic;
2 using Platform.Interfaces;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences.Converters
7 {
8     public class SequenceToItsLocalElementLevelsConverter<TLink> : LinksOperatorBase<TLink>,
        ↪ IConverter<IList<TLink>>
9     {
10         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
11
12         private readonly IConverter<Doublet<TLink>, TLink> _linkToItsFrequencyToNumberConveter;
13
14         public SequenceToItsLocalElementLevelsConverter(ILinks<TLink> links,
        ↪ IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter) : base(links)
        ↪ => _linkToItsFrequencyToNumberConveter = linkToItsFrequencyToNumberConveter;
15
16         public IList<TLink> Convert(IList<TLink> sequence)
17         {
18             var levels = new TLink[sequence.Count];
19             levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
20             for (var i = 1; i < sequence.Count - 1; i++)
21             {
22                 var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
23                 var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
24                 levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
25             }
26             levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
        ↪ sequence[sequence.Count - 1]);
27             return levels;
28         }
29
30         public TLink GetFrequencyNumber(TLink source, TLink target) =>
        ↪ _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLink>(source, target));
31     }
32 }

```

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

```

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

```

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

```

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

```

```

14
15     public MarkedSequenceCriterionMatcher(ILinks<TLink> links, TLink sequenceMarkerLink)
16     {
17         _links = links;
18         _sequenceMarkerLink = sequenceMarkerLink;
19     }
20
21     public bool IsMatched(TLink sequenceCandidate)
22     => _equalityComparer.Equals(_links.GetSource(sequenceCandidate), _sequenceMarkerLink)
23     || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarkerLink,
24         ↪ sequenceCandidate), _links.Constants.Null);
25 }

```

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

```

1  using System.Collections.Generic;
2  using Platform.Collections.Stacks;
3  using Platform.Data.Doublets.Sequences.HeightProviders;
4  using Platform.Data.Sequences;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Sequences
9  {
10     public class DefaultSequenceAppender<TLink> : LinksOperatorBase<TLink>,
11         ↪ ISequenceAppender<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ↪ EqualityComparer<TLink>.Default;
15
16         private readonly IStack<TLink> _stack;
17         private readonly ISequenceHeightProvider<TLink> _heightProvider;
18
19         public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
20             ↪ ISequenceHeightProvider<TLink> heightProvider)
21             : base(links)
22         {
23             _stack = stack;
24             _heightProvider = heightProvider;
25         }
26
27         public TLink Append(TLink sequence, TLink appendant)
28         {
29             var cursor = sequence;
30             while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
31             {
32                 var source = Links.GetSource(cursor);
33                 var target = Links.GetTarget(cursor);
34                 if (_equalityComparer.Equals(_heightProvider.Get(source),
35                     ↪ _heightProvider.Get(target)))
36                 {
37                     break;
38                 }
39                 else
40                 {
41                     _stack.Push(source);
42                     cursor = target;
43                 }
44             }
45             var left = cursor;
46             var right = appendant;
47             while (!_equalityComparer.Equals(cursor = _stack.Pop(), Links.Constants.Null))
48             {
49                 right = Links.GetOrCreate(left, right);
50                 left = cursor;
51             }
52             return Links.GetOrCreate(left, right);
53         }
54     }
55 }

```

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

```

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

```

```

9     public class DuplicateSegmentsCounter<TLink> : ICounter<int>
10     {
11         private readonly IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
            ↪ _duplicateFragmentsProvider;
12         public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLink>,
            ↪ IList<TLink>>>> duplicateFragmentsProvider) => _duplicateFragmentsProvider =
            ↪ duplicateFragmentsProvider;
13         public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
14     }
15 }

```

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

```

1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4  using Platform.Interfaces;
5  using Platform.Collections;
6  using Platform.Collections.Lists;
7  using Platform.Collections.Segments;
8  using Platform.Collections.Segments.Walkers;
9  using Platform.Singletons;
10 using Platform.Numbers;
11 using Platform.Data.Sequences;
12 using Platform.Data.Doublets.Unicode;
13
14 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16 namespace Platform.Data.Doublets.Sequences
17 {
18     public class DuplicateSegmentsProvider<TLink> :
            ↪ DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
            ↪ IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
19     {
20         private readonly ILinks<TLink> _links;
21         private readonly ISequences<TLink> _sequences;
22         private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
23         private BitString _visited;
24
25         private class ItemEqualityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
            ↪ IList<TLink>>>
26         {
27             private readonly IListEqualityComparer<TLink> _listComparer;
28             public ItemEqualityComparer() => _listComparer =
                ↪ Default<IListEqualityComparer<TLink>>.Instance;
29             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);
30             public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
                ↪ (_listComparer.GetHashCode(pair.Key),
                ↪ _listComparer.GetHashCode(pair.Value)).GetHashCode();
31         }
32
33         private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
34         {
35             private readonly IListComparer<TLink> _listComparer;
36
37             public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;
38
39             public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
                ↪ KeyValuePair<IList<TLink>, IList<TLink>> right)
40             {
41                 var intermediateResult = _listComparer.Compare(left.Key, right.Key);
42                 if (intermediateResult == 0)
43                 {
44                     intermediateResult = _listComparer.Compare(left.Value, right.Value);
45                 }
46                 return intermediateResult;
47             }
48         }
49
50         public DuplicateSegmentsProvider(ILinks<TLink> links, ISequences<TLink> sequences)
51             : base(minimumStringSegmentLength: 2)
52         {
53             _links = links;
54             _sequences = sequences;
55         }
56
57         public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
58         {

```

```

59     _groups = new HashSet<KeyValuePair<IList<TLink>,
    ↪     IList<TLink>>>(Default<ItemEqualityComparer>.Instance);
60     var count = _links.Count();
61     _visited = new BitString((long)(Integer<TLink>)count + 1);
62     _links.Each(link =>
63     {
64         var linkIndex = _links.GetIndex(link);
65         var linkBitIndex = (long)(Integer<TLink>)linkIndex;
66         if (!_visited.Get(linkBitIndex))
67         {
68             var sequenceElements = new List<TLink>();
69             _sequences.EachPart(sequenceElements.AddAndReturnTrue, 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(),
    ↪ duplicates));
97     }
98 }
99
100 private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
101 {
102     var duplicates = new List<TLink>();
103     var readAsElement = new HashSet<TLink>();
104     _sequences.Each(sequence =>
105     {
106         duplicates.Add(sequence);
107         readAsElement.Add(sequence);
108         return true; // Continue
109     }, segment);
110     if (duplicates.Any(x => _visited.Get((Integer<TLink>)x)))
111     {
112         return new List<TLink>();
113     }
114     foreach (var duplicate in duplicates)
115     {
116         var duplicateBitIndex = (long)(Integer<TLink>)duplicate;
117         _visited.Set(duplicateBitIndex);
118     }
119     if (_sequences is Sequences sequencesExperiments)
120     {
121         var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((H_
    ↪ ashSet<ulong>)(object)readAsElement,
    ↪ (IList<ulong>)segment);
122         foreach (var partiallyMatchedSequence in partiallyMatched)
123         {
124             TLink sequenceIndex = (Integer<TLink>)partiallyMatchedSequence;
125             duplicates.Add(sequenceIndex);
126         }
127     }
128     duplicates.Sort();
129     return duplicates;
130 }
131

```



```

132 private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
133 {
134     if (!(_links is ILinks<ulong> ulongLinks))
135     {
136         return;
137     }
138     var duplicatesKey = duplicatesItem.Key;
139     var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
140     Console.WriteLine($"{keyString} ({string.Join(", ", duplicatesKey)})");
141     var duplicatesList = duplicatesItem.Value;
142     for (int i = 0; i < duplicatesList.Count; i++)
143     {
144         ulong sequenceIndex = (Integer<TLink>)duplicatesList[i];
145         var formattedSequenceStructure = ulongLinks.FormatStructure(sequenceIndex, x =>
            ↳ Point<ulong>.IsPartialPoint(x), (sb, link) => _ =
            ↳ UnicodeMap.IsCharLink(link.Index) ?
            ↳ sb.Append(UnicodeMap.FromLinkToChar(link.Index)) : sb.Append(link.Index));
146         Console.WriteLine(formattedSequenceStructure);
147         var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
            ↳ ulongLinks);
148         Console.WriteLine(sequenceString);
149     }
150     Console.WriteLine();
151 }
152 }
153 }

```

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

```

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

```

```

49     }
50 }
51
52 [MethodImpl(MethodImplOptions.AggressiveInlining)]
53 public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
54 {
55     var doublet = new Doublet<TLink>(source, target);
56     return IncrementFrequency(ref doublet);
57 }
58
59 public void PrintFrequencies(IList<TLink> sequence)
60 {
61     for (var i = 1; i < sequence.Count; i++)
62     {
63         PrintFrequency(sequence[i - 1], sequence[i]);
64     }
65 }
66
67 public void PrintFrequency(TLink source, TLink target)
68 {
69     var number = GetFrequency(source, target).Frequency;
70     Console.WriteLine("{0},{1} - {2}", source, target, number);
71 }
72
73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
75 {
76     if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
77     {
78         data.IncrementFrequency();
79     }
80     else
81     {
82         var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
83         data = new LinkFrequency<TLink>(Integer<TLink>.One, link);
84         if (!_equalityComparer.Equals(link, default))
85         {
86             data.Frequency = Arithmetic.Add(data.Frequency,
87                 ↪ _frequencyCounter.Count(link));
88         }
89         _doubletsCache.Add(doublet, data);
90     }
91     return data;
92 }
93
94 public void ValidateFrequencies()
95 {
96     foreach (var entry in _doubletsCache)
97     {
98         var value = entry.Value;
99         var linkIndex = value.Link;
100         if (!_equalityComparer.Equals(linkIndex, default))
101         {
102             var frequency = value.Frequency;
103             var count = _frequencyCounter.Count(linkIndex);
104             // TODO: Why `frequency` always greater than `count` by 1?
105             if (((_comparer.Compare(frequency, count) > 0) &&
106                 ↪ (_comparer.Compare(Arithmetic.Subtract(frequency, count),
107                 ↪ Integer<TLink>.One) > 0))
108                 || ((_comparer.Compare(count, frequency) > 0) &&
109                 ↪ (_comparer.Compare(Arithmetic.Subtract(count, frequency),
110                 ↪ Integer<TLink>.One) > 0)))
111             {
112                 throw new InvalidOperationException("Frequencies validation failed.");
113             }
114         }
115         //else
116         // {
117         //     if (value.Frequency > 0)
118         //     {
119         //         var frequency = value.Frequency;
120         //         linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
121         //         var count = _countLinkFrequency(linkIndex);
122         //         if ((frequency > count && frequency - count > 1) || (count > frequency
123         //             ↪ && count - frequency > 1))
124         //             throw new Exception("Frequencies validation failed.");
125         //     }
126         // }

```

```

121         //}
122     }
123 }
124 }
125 }

```

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

```

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

```

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

```

1 using Platform.Interfaces;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
6 {
7     public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLink> :
8         ⇨ IConverter<Doublet<TLink>, TLink>
9     {
10         private readonly LinkFrequenciesCache<TLink> _cache;
11         public
12         ⇨ FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink>
13         ⇨ cache) => _cache = cache;
14         public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref source).Frequency;
15     }
16 }

```

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

```

1 using Platform.Interfaces;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
6 {
7     public class MarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
8         ⇨ SequenceSymbolFrequencyOneOffCounter<TLink>
9     {
10         private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
11
12         public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
13         ⇨ ICriterionMatcher<TLink> markedSequenceMatcher, TLink sequenceLink, TLink symbol)
14         : base(links, sequenceLink, symbol)
15         => _markedSequenceMatcher = markedSequenceMatcher;
16
17         public override TLink Count()
18         {
19             if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
20             {
21                 return default;
22             }
23             return base.Count();
24         }
25     }
26 }

```

```

22     }
23 }
24 }

```

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

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3  using Platform.Numbers;
4  using Platform.Data.Sequences;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
9  {
10     public class SequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13             ⇨ EqualityComparer<TLink>.Default;
14         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
15
16         protected readonly ILinks<TLink> _links;
17         protected readonly TLink _sequenceLink;
18         protected readonly TLink _symbol;
19         protected TLink _total;
20
21         public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink sequenceLink,
22             ⇨ TLink symbol)
23         {
24             _links = links;
25             _sequenceLink = sequenceLink;
26             _symbol = symbol;
27             _total = default;
28         }
29
30         public virtual TLink Count()
31         {
32             if (_comparer.Compare(_total, default) > 0)
33             {
34                 return _total;
35             }
36             StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
37                 ⇨ IsElement, VisitElement);
38             return _total;
39         }
40
41         private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
42             ⇨ _links.IsPartialPoint(x); // TODO: Use SequenceElementCriteriaMatcher instead of
43             ⇨ IsPartialPoint
44
45         private bool VisitElement(TLink element)
46         {
47             if (_equalityComparer.Equals(element, _symbol))
48             {
49                 _total = Arithmetic.Increment(_total);
50             }
51             return true;
52         }
53     }
54 }

```

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

```

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

```

```

18     public TLink Count(TLink argument) => new
        ↳ TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
        ↳ _markedSequenceMatcher, argument).Count();
19     }
20 }

```

./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> :
        ↳ TotalSequenceSymbolFrequencyOneOffCounter<TLink>
9     {
10         private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
11
12         public TotalMarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
        ↳ ICriterionMatcher<TLink> markedSequenceMatcher, TLink symbol)
13             : base(links, symbol)
14             => _markedSequenceMatcher = markedSequenceMatcher;
15
16         protected override void CountSequenceSymbolFrequency(TLink link)
17         {
18             var symbolFrequencyCounter = new
                ↳ MarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
                ↳ _markedSequenceMatcher, link, _symbol);
19             _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
20         }
21     }
22 }

```

./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
            ↳ TotalSequenceSymbolFrequencyOneOffCounter<TLink>(_links, symbol).Count();
12    }
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 =
            ↳ EqualityComparer<TLink>.Default;
12        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
13
14        protected readonly ILinks<TLink> _links;
15        protected readonly TLink _symbol;
16        protected readonly HashSet<TLink> _visits;
17        protected TLink _total;
18
19        public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
20        {
21            _links = links;
22            _symbol = symbol;
23            _visits = new HashSet<TLink>();
24            _total = default;
25        }
26
27        public TLink Count()
28        {

```

```

29         if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
30         {
31             return _total;
32         }
33         CountCore(_symbol);
34         return _total;
35     }
36
37     private void CountCore(TLink link)
38     {
39         var any = _links.Constants.Any;
40         if (_equalityComparer.Equals(_links.Count(any, link), default))
41         {
42             CountSequenceSymbolFrequency(link);
43         }
44         else
45         {
46             _links.Each(EachElementHandler, any, link);
47         }
48     }
49
50     protected virtual void CountSequenceSymbolFrequency(TLink link)
51     {
52         var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
53             ↪ link, _symbol);
54         _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
55     }
56
57     private TLink EachElementHandler(IList<TLink> doublet)
58     {
59         var constants = _links.Constants;
60         var doubletIndex = doublet[constants.IndexPart];
61         if (_visits.Add(doubletIndex))
62         {
63             CountCore(doubletIndex);
64         }
65         return constants.Continue;
66     }
67 }

```

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

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.HeightProviders
7  {
8      public class CachedSequenceHeightProvider<TLink> : LinksOperatorBase<TLink>,
9          ↪ ISequenceHeightProvider<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↪ EqualityComparer<TLink>.Default;
13
14         private readonly TLink _heightPropertyMarker;
15         private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
16         private readonly IConverter<TLink> _addressToUnaryNumberConverter;
17         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
18         private readonly IPropertiesOperator<TLink, TLink, TLink> _propertyOperator;
19
20         public CachedSequenceHeightProvider(
21             ILinks<TLink> links,
22             ISequenceHeightProvider<TLink> baseHeightProvider,
23             IConverter<TLink> addressToUnaryNumberConverter,
24             IConverter<TLink> unaryNumberToAddressConverter,
25             TLink heightPropertyMarker,
26             IPropertiesOperator<TLink, TLink, TLink> propertyOperator)
27             : base(links)
28         {
29             _heightPropertyMarker = heightPropertyMarker;
30             _baseHeightProvider = baseHeightProvider;
31             _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
32             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
33             _propertyOperator = propertyOperator;
34         }
35
36         public TLink Get(TLink sequence)
37         {
38             TLink height;
39             var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);

```

```

38         if (_equalityComparer.Equals(heightValue, default))
39         {
40             height = _baseHeightProvider.Get(sequence);
41             heightValue = _addressToUnaryNumberConverter.Convert(height);
42             _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
43         }
44         else
45         {
46             height = _unaryNumberToAddressConverter.Convert(heightValue);
47         }
48         return height;
49     }
50 }
51 }

```

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

```

1  using Platform.Interfaces;
2  using Platform.Numbers;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.HeightProviders
7  {
8      public class DefaultSequenceRightHeightProvider<TLink> : LinksOperatorBase<TLink>,
9          ↳ ISequenceHeightProvider<TLink>
10     {
11         private readonly ICriterionMatcher<TLink> _elementMatcher;
12
13         public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriterionMatcher<TLink>
14             ↳ elementMatcher) : base(links) => _elementMatcher = elementMatcher;
15
16         public TLink Get(TLink sequence)
17         {
18             var height = default(TLink);
19             var pairOrElement = sequence;
20             while (!_elementMatcher.IsMatched(pairOrElement))
21             {
22                 pairOrElement = Links.GetTarget(pairOrElement);
23                 height = Arithmetic.Increment(height);
24             }
25             return height;
26         }
27     }
28 }

```

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

```

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

```

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

```

1  using System.Collections.Generic;
2  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.Indexes
7  {
8      public class CachedFrequencyIncrementingSequenceIndex<TLink> : ISequenceIndex<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ↳ EqualityComparer<TLink>.Default;
12
13         private readonly LinkFrequenciesCache<TLink> _cache;
14
15         public CachedFrequencyIncrementingSequenceIndex(LinkFrequenciesCache<TLink> cache) =>
16             ↳ _cache = cache;
17
18         public bool Add(IList<TLink> sequence)
19         {
20             var indexed = true;
21             var i = sequence.Count;

```

```

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
43     public bool MightContain(IList<TLink> sequence)
44     {
45         var indexed = true;
46         var i = sequence.Count;
47         while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
48         return indexed;
49     }
50
51     private bool IsIndexed(TLink source, TLink target)
52     {
53         var frequency = _cache.GetFrequency(source, target);
54         if (frequency == null)
55         {
56             return false;
57         }
58         return !_equalityComparer.Equals(frequency.Frequency, default);
59     }
60 }
61 }

```

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

```

1  using Platform.Interfaces;
2  using System.Collections.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.Indexes
7  {
8      public class FrequencyIncrementingSequenceIndex<TLink> : SequenceIndex<TLink>,
9          ↪ ISequenceIndex<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↪ EqualityComparer<TLink>.Default;
13
14         private readonly IPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
15         private readonly IIncrementer<TLink> _frequencyIncrementer;
16
17         public FrequencyIncrementingSequenceIndex(IList<TLink> links, IPropertyOperator<TLink,
18             ↪ TLink> frequencyPropertyOperator, IIncrementer<TLink> frequencyIncrementer)
19             : base(links)
20         {
21             _frequencyPropertyOperator = frequencyPropertyOperator;
22             _frequencyIncrementer = frequencyIncrementer;
23         }
24
25         public override bool Add(IList<TLink> sequence)
26         {
27             var indexed = true;
28             var i = sequence.Count;
29             while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
30                 ↪ { }
31             for (; i >= 1; i--)
32             {
33                 Increment(Links.GetOrCreate(sequence[i - 1], sequence[i]));
34             }
35             return indexed;
36         }
37     }
38 }

```



```

32     }
33
34     private bool IsIndexedWithIncrement(TLink source, TLink target)
35     {
36         var link = Links.SearchOrDefault(source, target);
37         var indexed = !_equalityComparer.Equals(link, default);
38         if (indexed)
39         {
40             Increment(link);
41         }
42         return indexed;
43     }
44
45     private void Increment(TLink link)
46     {
47         var previousFrequency = _frequencyPropertyOperator.Get(link);
48         var frequency = _frequencyIncrementer.Increment(previousFrequency);
49         _frequencyPropertyOperator.Set(link, frequency);
50     }
51 }
52 }

```

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

```

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

```

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

```

1  using System.Collections.Generic;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Sequences.Indexes
6  {
7      public class SequenceIndex<TLink> : LinksOperatorBase<TLink>, ISequenceIndex<TLink>
8      {
9          private static readonly EqualityComparer<TLink> _equalityComparer =
10             ↪ EqualityComparer<TLink>.Default;
11
12         public SequenceIndex(ILinks<TLink> links) : base(links) { }
13
14         public virtual bool Add(IList<TLink> sequence)
15         {
16             var indexed = true;
17             var i = sequence.Count;
18             while (--i >= 1 && (indexed =
19                 ↪ !_equalityComparer.Equals(Links.SearchOrDefault(sequence[i - 1], sequence[i]),
20                 ↪ default))) { }
21             for (; i >= 1; i--)
22             {
23                 Links.GetOrCreate(sequence[i - 1], sequence[i]);
24             }
25             return indexed;
26         }
27
28         public virtual bool MightContain(IList<TLink> sequence)
29         {
30             var indexed = true;
31             var i = sequence.Count;
32             while (--i >= 1 && (indexed =
33                 ↪ !_equalityComparer.Equals(Links.SearchOrDefault(sequence[i - 1], sequence[i]),
34                 ↪ default))) { }
35             return indexed;
36         }
37     }
38 }

```

```

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/Sequences.cs
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections;
6  using Platform.Collections.Lists;
7  using Platform.Threading.Synchronization;
8  using Platform.Singletons;
9  using LinkIndex = System.UInt64;
10 using Platform.Data.Constants;
11 using Platform.Data.Sequences;
12 using Platform.Data.Doublets.Sequences.Walkers;
13 using Platform.Collections.Stacks;
14
15 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
16
17 namespace Platform.Data.Doublets.Sequences
18 {
19     /// <summary>
20     /// Представляет коллекцию последовательностей связей.

```

```

21 /// </summary>
22 /// <remarks>
23 /// Обязательно реализовать атомарность каждого публичного метода.
24 ///
25 /// TODO:
26 ///
27 /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
28 /// через естественную группировку по unicode типам, все whitespace вместе, все символы
29 /// → вместе, все числа вместе и т.п.
30 /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
31 /// → графа)
32 ///
33 /// х*у - найти все связи между, в последовательностях любой формы, если не стоит
34 /// → ограничитель на то, что является последовательностью, а что нет,
35 /// то находятся любые структуры связей, которые содержат эти элементы именно в таком
36 /// → порядке.
37 ///
38 /// Рост последовательности слева и справа.
39 /// Поиск со звёздочкой.
40 /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
41 /// так же проблема может быть решена при реализации дистанционных триггеров.
42 /// Нужны ли уникальные указатели вообще?
43 /// Что если обращение к информации будет происходить через содержимое всегда?
44 ///
45 /// Писать тесты.
46 ///
47 /// Можно убрать зависимость от конкретной реализации Links,
48 /// на зависимость от абстрактного элемента, который может быть представлен несколькими
49 /// → способами.
50 ///
51 /// Можно ли как-то сделать один общий интерфейс
52 ///
53 ///
54 /// Блокчейн и/или гит для распределённой записи транзакций.
55 ///
56 /// </remarks>
57 public partial class Sequences : ISequences<ulong> // IList<string>, IList<ulong[]> (после
58 → завершения реализации Sequences)
59 {
60     private static readonly LinksCombinedConstants<bool, ulong, long> _constants =
61     → Default<LinksCombinedConstants<bool, ulong, long>>.Instance;
62
63     /// <summary>Возвращает значение ulong, обозначающее любое количество связей.</summary>
64     public const ulong ZeroOrMany = ulong.MaxValue;
65
66     public SequencesOptions<ulong> Options;
67     public readonly SynchronizedLinks<ulong> Links;
68     public readonly ISynchronization Sync;
69
70     public Sequences(SynchronizedLinks<ulong> links)
71     : this(links, new SequencesOptions<ulong>())
72     {
73     }
74
75     public Sequences(SynchronizedLinks<ulong> links, SequencesOptions<ulong> options)
76     {
77         Links = links;
78         Sync = links.SyncRoot;
79         Options = options;
80
81         Options.ValidateOptions();
82         Options.InitOptions(Links);
83     }
84
85     public bool IsSequence(ulong sequence)
86     {
87         return Sync.ExecuteReadOperation(() =>
88         {
89             if (Options.UseSequenceMarker)
90             {
91                 return Options.MarkedSequenceMatcher.IsMatched(sequence);
92             }
93             return !Links.Unsync.IsPartialPoint(sequence);
94         });
95     }
96
97     [MethodImpl(MethodImplOptions.AggressiveInlining)]
98     private ulong GetSequenceByElements(ulong sequence)

```

```

93     {
94         if (Options.UseSequenceMarker)
95         {
96             return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
97         }
98         return sequence;
99     }
100
101 private ulong GetSequenceElements(ulong sequence)
102 {
103     if (Options.UseSequenceMarker)
104     {
105         var linkContents = new UInt64Link(Links.GetLink(sequence));
106         if (linkContents.Source == Options.SequenceMarkerLink)
107         {
108             return linkContents.Target;
109         }
110         if (linkContents.Target == Options.SequenceMarkerLink)
111         {
112             return linkContents.Source;
113         }
114     }
115     return sequence;
116 }
117
118 #region Count
119
120 public ulong Count(params ulong[] sequence)
121 {
122     if (sequence.Length == 0)
123     {
124         return Links.Count(_constants.Any, Options.SequenceMarkerLink, _constants.Any);
125     }
126     if (sequence.Length == 1) // Первая связь это адрес
127     {
128         if (sequence[0] == _constants.Null)
129         {
130             return 0;
131         }
132         if (sequence[0] == _constants.Any)
133         {
134             return Count();
135         }
136         if (Options.UseSequenceMarker)
137         {
138             return Links.Count(_constants.Any, Options.SequenceMarkerLink, sequence[0]);
139         }
140         return Links.Exists(sequence[0]) ? 1UL : 0;
141     }
142     throw new NotImplementedException();
143 }
144
145 private ulong CountUsages(params ulong[] restrictions)
146 {
147     if (restrictions.Length == 0)
148     {
149         return 0;
150     }
151     if (restrictions.Length == 1) // Первая связь это адрес
152     {
153         if (restrictions[0] == _constants.Null)
154         {
155             return 0;
156         }
157         if (Options.UseSequenceMarker)
158         {
159             var elementsLink = GetSequenceElements(restrictions[0]);
160             var sequenceLink = GetSequenceByElements(elementsLink);
161             if (sequenceLink != _constants.Null)
162             {
163                 return Links.Count(sequenceLink) + Links.Count(elementsLink) - 1;
164             }
165             return Links.Count(elementsLink);
166         }
167         return Links.Count(restrictions[0]);
168     }
169     throw new NotImplementedException();
170 }
171

```

```

172 #endregion
173
174 #region Create
175
176 public ulong Create(params ulong[] sequence)
177 {
178     return Sync.ExecuteWriteOperation(() =>
179     {
180         if (sequence.IsNullOrEmpty())
181         {
182             return _constants.Null;
183         }
184         Links.EnsureEachLinkExists(sequence);
185         return CreateCore(sequence);
186     });
187 }
188
189 private ulong CreateCore(params ulong[] sequence)
190 {
191     if (Options.UseIndex)
192     {
193         Options.Index.Add(sequence);
194     }
195     var sequenceRoot = default(ulong);
196     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
197     {
198         var matches = Each(sequence);
199         if (matches.Count > 0)
200         {
201             sequenceRoot = matches[0];
202         }
203     }
204     else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
205     {
206         return CompactCore(sequence);
207     }
208     if (sequenceRoot == default)
209     {
210         sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
211     }
212     if (Options.UseSequenceMarker)
213     {
214         Links.Unsync.CreateAndUpdate(Options.SequenceMarkerLink, sequenceRoot);
215     }
216     return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
217 }
218
219 #endregion
220
221 #region Each
222
223 public List<ulong> Each(params ulong[] sequence)
224 {
225     var results = new List<ulong>();
226     Each(results.AddAndReturnTrue, sequence);
227     return results;
228 }
229
230 public bool Each(Func<ulong, bool> handler, IList<ulong> sequence)
231 {
232     return Sync.ExecuteReadOperation(() =>
233     {
234         if (sequence.IsNullOrEmpty())
235         {
236             return true;
237         }
238         Links.EnsureEachLinkIsAnyOrExists(sequence);
239         if (sequence.Count == 1)
240         {
241             var link = sequence[0];
242             if (link == _constants.Any)
243             {
244                 return Links.Unsync.Each(_constants.Any, _constants.Any, handler);
245             }
246             return handler(link);
247         }
248         if (sequence.Count == 2)
249         {
250             return Links.Unsync.Each(sequence[0], sequence[1], handler);
251         }
252     });
253 }

```

```

251     }
252     if (Options.UseIndex && !Options.Index.MightContain(sequence))
253     {
254         return false;
255     }
256     return EachCore(handler, sequence);
257 });
258 }
259
260 private bool EachCore(Func<ulong, bool> handler, IList<ulong> sequence)
261 {
262     var matcher = new Matcher(this, sequence, new HashSet<LinkIndex>(), handler);
263     // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
264     ↪ Id.
265     Func<ulong, bool> innerHandler = Options.UseSequenceMarker ? (Func<ulong,
266     ↪ bool>)matcher.HandleFullMatchedSequence : matcher.HandleFullMatched;
267     //if (sequence.Length >= 2)
268     if (!StepRight(innerHandler, sequence[0], sequence[1]))
269     {
270         return false;
271     }
272     var last = sequence.Count - 2;
273     for (var i = 1; i < last; i++)
274     {
275         if (!PartialStepRight(innerHandler, sequence[i], sequence[i + 1]))
276         {
277             return false;
278         }
279     }
280     if (sequence.Count >= 3)
281     {
282         if (!StepLeft(innerHandler, sequence[sequence.Count - 2],
283         ↪ sequence[sequence.Count - 1]))
284         {
285             return false;
286         }
287     }
288     return true;
289 }
290
291 private bool PartialStepRight(Func<ulong, bool> handler, ulong left, ulong right)
292 {
293     return Links.Unsync.Each(_constants.Any, left, doublet =>
294     {
295         if (!StepRight(handler, doublet, right))
296         {
297             return false;
298         }
299         if (left != doublet)
300         {
301             return PartialStepRight(handler, doublet, right);
302         }
303     });
304     return true;
305 }
306
307 private bool StepRight(Func<ulong, bool> handler, ulong left, ulong right) =>
308     ↪ Links.Unsync.Each(left, _constants.Any, rightStep => TryStepRightUp(handler, right,
309     ↪ rightStep));
310
311 private bool TryStepRightUp(Func<ulong, bool> handler, ulong right, ulong stepFrom)
312 {
313     var upStep = stepFrom;
314     var firstSource = Links.Unsync.GetTarget(upStep);
315     while (firstSource != right && firstSource != upStep)
316     {
317         upStep = firstSource;
318         firstSource = Links.Unsync.GetSource(upStep);
319     }
320     if (firstSource == right)
321     {
322         return handler(stepFrom);
323     }
324     return true;
325 }
326
327 private bool StepLeft(Func<ulong, bool> handler, ulong left, ulong right) =>
328     ↪ Links.Unsync.Each(_constants.Any, right, leftStep => TryStepLeftUp(handler, left,
329     ↪ leftStep));

```

```

323
324 private bool TryStepLeftUp(Func<ulong, bool> handler, ulong left, ulong stepFrom)
325 {
326     var upStep = stepFrom;
327     var firstTarget = Links.Unsync.GetSource(upStep);
328     while (firstTarget != left && firstTarget != upStep)
329     {
330         upStep = firstTarget;
331         firstTarget = Links.Unsync.GetTarget(upStep);
332     }
333     if (firstTarget == left)
334     {
335         return handler(stepFrom);
336     }
337     return true;
338 }
339
340 #endregion
341
342 #region Update
343
344 public ulong Update(ulong[] sequence, ulong[] newSequence)
345 {
346     if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
347     {
348         return _constants.Null;
349     }
350     if (sequence.IsNullOrEmpty())
351     {
352         return Create(newSequence);
353     }
354     if (newSequence.IsNullOrEmpty())
355     {
356         Delete(sequence);
357         return _constants.Null;
358     }
359     return Sync.ExecuteWriteOperation(() =>
360     {
361         Links.EnsureEachLinkIsAnyOrExists(sequence);
362         Links.EnsureEachLinkExists(newSequence);
363         return UpdateCore(sequence, newSequence);
364     });
365 }
366
367 private ulong UpdateCore(ulong[] sequence, ulong[] newSequence)
368 {
369     ulong bestVariant;
370     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
371         ↪ !sequence.EqualTo(newSequence))
372     {
373         bestVariant = CompactCore(newSequence);
374     }
375     else
376     {
377         bestVariant = CreateCore(newSequence);
378     }
379     // TODO: Check all options only ones before loop execution
380     // Возможно нужно две версии Each, возвращающий фактические последовательности и с
381     ↪ маркером,
382     // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
383     ↪ можно получить имея только фактические последовательности.
384     foreach (var variant in Each(sequence))
385     {
386         if (variant != bestVariant)
387         {
388             UpdateOneCore(variant, bestVariant);
389         }
390     }
391     return bestVariant;
392 }
393
394 private void UpdateOneCore(ulong sequence, ulong newSequence)
395 {
396     if (Options.UseGarbageCollection)
397     {
398         var sequenceElements = GetSequenceElements(sequence);
399         var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
400         var sequenceLink = GetSequenceByElements(sequenceElements);
401         var newSequenceElements = GetSequenceElements(newSequence);

```

```

399     var newSequenceLink = GetSequenceByElements(newSequenceElements);
400     if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
401     {
402         if (sequenceLink != _constants.Null)
403         {
404             Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
405         }
406         Links.Unsync.MergeUsages(sequenceElements, newSequenceElements);
407     }
408     ClearGarbage(sequenceElementsContents.Source);
409     ClearGarbage(sequenceElementsContents.Target);
410 }
411 else
412 {
413     if (Options.UseSequenceMarker)
414     {
415         var sequenceElements = GetSequenceElements(sequence);
416         var sequenceLink = GetSequenceByElements(sequenceElements);
417         var newSequenceElements = GetSequenceElements(newSequence);
418         var newSequenceLink = GetSequenceByElements(newSequenceElements);
419         if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
420         {
421             if (sequenceLink != _constants.Null)
422             {
423                 Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
424             }
425             Links.Unsync.MergeUsages(sequenceElements, newSequenceElements);
426         }
427     }
428     else
429     {
430         if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
431         {
432             Links.Unsync.MergeUsages(sequence, newSequence);
433         }
434     }
435 }
436 }
437
438 #endregion
439
440 #region Delete
441
442 public void Delete(params ulong[] sequence)
443 {
444     Sync.ExecuteWriteOperation(() =>
445     {
446         // TODO: Check all options only ones before loop execution
447         foreach (var linkToDelete in Each(sequence))
448         {
449             DeleteOneCore(linkToDelete);
450         }
451     });
452 }
453
454 private void DeleteOneCore(ulong link)
455 {
456     if (Options.UseGarbageCollection)
457     {
458         var sequenceElements = GetSequenceElements(link);
459         var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
460         var sequenceLink = GetSequenceByElements(sequenceElements);
461         if (Options.UseCascadeDelete || CountUsages(link) == 0)
462         {
463             if (sequenceLink != _constants.Null)
464             {
465                 Links.Unsync.Delete(sequenceLink);
466             }
467             Links.Unsync.Delete(link);
468         }
469         ClearGarbage(sequenceElementsContents.Source);
470         ClearGarbage(sequenceElementsContents.Target);
471     }
472     else
473     {
474         if (Options.UseSequenceMarker)
475         {
476             var sequenceElements = GetSequenceElements(link);

```



```

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

```

```

554         foreach (var part in Options.Walker.Walk(sequence))
555         {
556             if (!handler(part))
557             {
558                 return false;
559             }
560         }
561         return true;
562     });
563 }
564
565 public class Matcher : RightSequenceWalker

```

```

629     {
630         return _stopableHandler(sequenceToMatch);
631     }
632     return true;
633 }
634
635 public bool HandleFullMatchedSequence(ulong sequenceToMatch)
636 {
637     var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
638     if (sequence != _constants.Null && FullMatch(sequenceToMatch) &&
        ↪ _results.Add(sequenceToMatch))
639     {
640         return _stopableHandler(sequence);
641     }
642     return true;
643 }
644
645 /// <remarks>
646 /// TODO: Add support for LinksConstants.Any
647 /// </remarks>
648 public bool PartialMatch(LinkIndex sequenceToMatch)
649 {
650     _filterPosition = -1;
651     foreach (var part in Walk(sequenceToMatch))
652     {
653         if (!PartialMatchCore(part))
654         {
655             break;
656         }
657     }
658     return _filterPosition == _patternSequence.Count - 1;
659 }
660
661 private bool PartialMatchCore(LinkIndex element)
662 {
663     if (_filterPosition == (_patternSequence.Count - 1))
664     {
665         return false; // Нашлось
666     }
667     if (_filterPosition >= 0)
668     {
669         if (element == _patternSequence[_filterPosition + 1])
670         {
671             _filterPosition++;
672         }
673         else
674         {
675             _filterPosition = -1;
676         }
677     }
678     if (_filterPosition < 0)
679     {
680         if (element == _patternSequence[0])
681         {
682             _filterPosition = 0;
683         }
684     }
685     return true; // Ищем дальше
686 }
687
688 public void AddPartialMatchedToResults(ulong sequenceToMatch)
689 {
690     if (PartialMatch(sequenceToMatch))
691     {
692         _results.Add(sequenceToMatch);
693     }
694 }
695
696 public bool HandlePartialMatched(ulong sequenceToMatch)
697 {
698     if (PartialMatch(sequenceToMatch))
699     {
700         return _stopableHandler(sequenceToMatch);
701     }
702     return true;
703 }
704
705 public void AddAllPartialMatchedToResults(IEnumerable<ulong> sequencesToMatch)
706 {

```

```

707         foreach (var sequenceToMatch in sequencesToMatch)
708         {
709             if (PartialMatch(sequenceToMatch))
710             {
711                 _results.Add(sequenceToMatch);
712             }
713         }
714     }
715
716     public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<ulong>
↵ sequencesToMatch)
717     {
718         foreach (var sequenceToMatch in sequencesToMatch)
719         {
720             if (PartialMatch(sequenceToMatch))
721             {
722                 _readAsElements.Add(sequenceToMatch);
723                 _results.Add(sequenceToMatch);
724             }
725         }
726     }
727 }
728
729 #endregion
730 }
731 }

```

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

```

1  using System;
2  using LinkIndex = System.UInt64;
3  using System.Collections.Generic;
4  using Stack = System.Collections.Generic.Stack<ulong>;
5  using System.Linq;
6  using System.Text;
7  using Platform.Collections;
8  using Platform.Data.Exceptions;
9  using Platform.Data.Sequences;
10 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
11 using Platform.Data.Doublets.Sequences.Walkers;
12 using Platform.Collections.Stacks;
13
14 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16 namespace Platform.Data.Doublets.Sequences
17 {
18     partial class Sequences
19     {
20         #region Create All Variants (Not Practical)
21
22         /// <remarks>
23         /// Number of links that is needed to generate all variants for
24         /// sequence of length N corresponds to https://oeis.org/A014143/list sequence.
25         /// </remarks>
26         public ulong[] CreateAllVariants2(ulong[] sequence)
27         {
28             return Sync.ExecuteWriteOperation(() =>
29             {
30                 if (sequence.IsNullOrEmpty())
31                 {
32                     return new ulong[0];
33                 }
34                 Links.EnsureEachLinkExists(sequence);
35                 if (sequence.Length == 1)
36                 {
37                     return sequence;
38                 }
39                 return CreateAllVariants2Core(sequence, 0, sequence.Length - 1);
40             });
41         }
42
43         private ulong[] CreateAllVariants2Core(ulong[] sequence, long startAt, long stopAt)
44         {
45             #if DEBUG
46                 if ((stopAt - startAt) < 0)
47                 {
48                     throw new ArgumentOutOfRangeException(nameof(startAt), "startAt должен быть
↵ меньше или равен stopAt");
49                 }
50             #endif
51             if ((stopAt - startAt) == 0)

```

```

52     {
53         return new[] { sequence[startAt] };
54     }
55     if ((stopAt - startAt) == 1)
56     {
57         return new[] { Links.Unsync.CreateAndUpdate(sequence[startAt], sequence[stopAt])
58             ↪ };
59     }
60     var variants = new ulong[(ulong)Platform.Numbers.Math.Catalan(stopAt - startAt)];
61     var last = 0;
62     for (var splitter = startAt; splitter < stopAt; splitter++)
63     {
64         var left = CreateAllVariants2Core(sequence, startAt, splitter);
65         var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
66         for (var i = 0; i < left.Length; i++)
67         {
68             for (var j = 0; j < right.Length; j++)
69             {
70                 var variant = Links.Unsync.CreateAndUpdate(left[i], right[j]);
71                 if (variant == _constants.Null)
72                 {
73                     throw new NotImplementedException("Creation cancellation is not
74                         ↪ implemented.");
75                 }
76                 variants[last++] = variant;
77             }
78         }
79     }
80     return variants;
81 }
82 public List<ulong> CreateAllVariants1(params ulong[] sequence)
83 {
84     return Sync.ExecuteWriteOperation(() =>
85     {
86         if (sequence.IsNullOrEmpty())
87         {
88             return new List<ulong>();
89         }
90         Links.Unsync.EnsureEachLinkExists(sequence);
91         if (sequence.Length == 1)
92         {
93             return new List<ulong> { sequence[0] };
94         }
95         var results = new
96             ↪ List<ulong>((int)Platform.Numbers.Math.Catalan(sequence.Length));
97         return CreateAllVariants1Core(sequence, results);
98     });
99 }
100 private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
101 {
102     if (sequence.Length == 2)
103     {
104         var link = Links.Unsync.CreateAndUpdate(sequence[0], sequence[1]);
105         if (link == _constants.Null)
106         {
107             throw new NotImplementedException("Creation cancellation is not
108                 ↪ implemented.");
109         }
110         results.Add(link);
111         return results;
112     }
113     var innerSequenceLength = sequence.Length - 1;
114     var innerSequence = new ulong[innerSequenceLength];
115     for (var li = 0; li < innerSequenceLength; li++)
116     {
117         var link = Links.Unsync.CreateAndUpdate(sequence[li], sequence[li + 1]);
118         if (link == _constants.Null)
119         {
120             throw new NotImplementedException("Creation cancellation is not
121                 ↪ implemented.");
122         }
123         for (var isi = 0; isi < li; isi++)
124         {
125             innerSequence[isi] = sequence[isi];
126         }
127         innerSequence[li] = link;

```

```

125         for (var isi = li + 1; isi < innerSequenceLength; isi++)
126         {
127             innerSequence[isi] = sequence[isi + 1];
128         }
129         CreateAllVariants1Core(innerSequence, results);
130     }
131     return results;
132 }
133
134 #endregion
135
136 public HashSet<ulong> Each1(params ulong[] sequence)
137 {
138     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
139     Each1(link =>
140     {
141         if (!visitedLinks.Contains(link))
142         {
143             visitedLinks.Add(link); // изучить почему случаются повторы
144         }
145         return true;
146     }, sequence);
147     return visitedLinks;
148 }
149
150 private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
151 {
152     if (sequence.Length == 2)
153     {
154         Links.Unsync.Each(sequence[0], sequence[1], handler);
155     }
156     else
157     {
158         var innerSequenceLength = sequence.Length - 1;
159         for (var li = 0; li < innerSequenceLength; li++)
160         {
161             var left = sequence[li];
162             var right = sequence[li + 1];
163             if (left == 0 && right == 0)
164             {
165                 continue;
166             }
167             var linkIndex = li;
168             ulong[] innerSequence = null;
169             Links.Unsync.Each(left, right, doublet =>
170             {
171                 if (innerSequence == null)
172                 {
173                     innerSequence = new ulong[innerSequenceLength];
174                     for (var isi = 0; isi < linkIndex; isi++)
175                     {
176                         innerSequence[isi] = sequence[isi];
177                     }
178                     for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
179                     {
180                         innerSequence[isi] = sequence[isi + 1];
181                     }
182                 }
183                 innerSequence[linkIndex] = doublet;
184                 Each1(handler, innerSequence);
185                 return _constants.Continue;
186             });
187         }
188     }
189 }
190
191 public HashSet<ulong> EachPart(params ulong[] sequence)
192 {
193     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
194     EachPartCore(link =>
195     {
196         if (!visitedLinks.Contains(link))
197         {
198             visitedLinks.Add(link); // изучить почему случаются повторы
199         }
200         return true;
201     }, sequence);
202     return visitedLinks;
203 }

```

```

204 public void EachPart(Func<ulong, bool> handler, params ulong[] sequence)
205 {
206     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
207     EachPartCore(link =>
208     {
209         if (!visitedLinks.Contains(link))
210         {
211             visitedLinks.Add(link); // изучить почему случаются повторы
212             return handler(link);
213         }
214         return true;
215     }, sequence);
216 }
217
218 private void EachPartCore(Func<ulong, bool> handler, params ulong[] sequence)
219 {
220     if (sequence.IsNullOrEmpty())
221     {
222         return;
223     }
224     Links.EnsureEachLinkIsAnyOrExists(sequence);
225     if (sequence.Length == 1)
226     {
227         var link = sequence[0];
228         if (link > 0)
229         {
230             handler(link);
231         }
232         else
233         {
234             Links.Each(_constants.Any, _constants.Any, handler);
235         }
236     }
237     else if (sequence.Length == 2)
238     {
239         //_links.Each(sequence[0], sequence[1], handler);
240         //  o_|      x_o ...
241         // x_|      |___|
242         Links.Each(sequence[1], _constants.Any, doublet =>
243         {
244             var match = Links.SearchOrDefault(sequence[0], doublet);
245             if (match != _constants.Null)
246             {
247                 handler(match);
248             }
249             return true;
250         });
251         // |_x      ... x_o
252         // |_o      |___|
253         Links.Each(_constants.Any, sequence[0], doublet =>
254         {
255             var match = Links.SearchOrDefault(doublet, sequence[1]);
256             if (match != 0)
257             {
258                 handler(match);
259             }
260             return true;
261         });
262         //          . _x o _ .
263         //          |___|
264         PartialStepRight(x => handler(x), sequence[0], sequence[1]);
265     }
266     else
267     {
268         // TODO: Implement other variants
269         return;
270     }
271 }
272
273 private void PartialStepRight(Action<ulong> handler, ulong left, ulong right)
274 {
275     Links.Unsync.Each(_constants.Any, left, doublet =>
276     {
277         StepRight(handler, doublet, right);
278         if (left != doublet)
279         {
280             PartialStepRight(handler, doublet, right);
281         }
282     });
283 }

```

```

282     }
283     return true;
284 });
285 }
286
287 private void StepRight(Action<ulong> handler, ulong left, ulong right)
288 {
289     Links.Unsync.Each(left, _constants.Any, rightStep =>
290     {
291         TryStepRightUp(handler, right, rightStep);
292         return true;
293     });
294 }
295
296 private void TryStepRightUp(Action<ulong> handler, ulong right, ulong stepFrom)
297 {
298     var upStep = stepFrom;
299     var firstSource = Links.Unsync.GetTarget(upStep);
300     while (firstSource != right && firstSource != upStep)
301     {
302         upStep = firstSource;
303         firstSource = Links.Unsync.GetSource(upStep);
304     }
305     if (firstSource == right)
306     {
307         handler(stepFrom);
308     }
309 }
310
311 // TODO: Test
312 private void PartialStepLeft(Action<ulong> handler, ulong left, ulong right)
313 {
314     Links.Unsync.Each(right, _constants.Any, doublet =>
315     {
316         StepLeft(handler, left, doublet);
317         if (right != doublet)
318         {
319             PartialStepLeft(handler, left, doublet);
320         }
321         return true;
322     });
323 }
324
325 private void StepLeft(Action<ulong> handler, ulong left, ulong right)
326 {
327     Links.Unsync.Each(_constants.Any, right, leftStep =>
328     {
329         TryStepLeftUp(handler, left, leftStep);
330         return true;
331     });
332 }
333
334 private void TryStepLeftUp(Action<ulong> handler, ulong left, ulong stepFrom)
335 {
336     var upStep = stepFrom;
337     var firstTarget = Links.Unsync.GetSource(upStep);
338     while (firstTarget != left && firstTarget != upStep)
339     {
340         upStep = firstTarget;
341         firstTarget = Links.Unsync.GetTarget(upStep);
342     }
343     if (firstTarget == left)
344     {
345         handler(stepFrom);
346     }
347 }
348
349 private bool StartsWith(ulong sequence, ulong link)
350 {
351     var upStep = sequence;
352     var firstSource = Links.Unsync.GetSource(upStep);
353     while (firstSource != link && firstSource != upStep)
354     {
355         upStep = firstSource;
356         firstSource = Links.Unsync.GetSource(upStep);
357     }
358     return firstSource == link;
359 }
360

```



```

361 private bool EndsWith(ulong sequence, ulong link)
362 {
363     var upStep = sequence;
364     var lastTarget = Links.Unsync.GetTarget(upStep);
365     while (lastTarget != link && lastTarget != upStep)
366     {
367         upStep = lastTarget;
368         lastTarget = Links.Unsync.GetTarget(upStep);
369     }
370     return lastTarget == link;
371 }
372
373 public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
374 {
375     return Sync.ExecuteReadOperation(() =>
376     {
377         var results = new List<ulong>();
378         if (sequence.Length > 0)
379         {
380             Links.EnsureEachLinkExists(sequence);
381             var firstElement = sequence[0];
382             if (sequence.Length == 1)
383             {
384                 results.Add(firstElement);
385                 return results;
386             }
387             if (sequence.Length == 2)
388             {
389                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
390                 if (doublet != _constants.Null)
391                 {
392                     results.Add(doublet);
393                 }
394                 return results;
395             }
396             var linksInSequence = new HashSet<ulong>(sequence);
397             void handler(ulong result)
398             {
399                 var filterPosition = 0;
400                 StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
401                     ↪ Links.Unsync.GetTarget,
402                     ↪ x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
403                     ↪ x =>
404                     {
405                         if (filterPosition == sequence.Length)
406                         {
407                             filterPosition = -2; // Длиннее чем нужно
408                             return false;
409                         }
410                         if (x != sequence[filterPosition])
411                         {
412                             filterPosition = -1;
413                             return false; // Начинается иначе
414                         }
415                         filterPosition++;
416                         return true;
417                     }
418                 });
419                 if (filterPosition == sequence.Length)
420                 {
421                     results.Add(result);
422                 }
423             }
424             if (sequence.Length >= 2)
425             {
426                 StepRight(handler, sequence[0], sequence[1]);
427             }
428             var last = sequence.Length - 2;
429             for (var i = 1; i < last; i++)
430             {
431                 PartialStepRight(handler, sequence[i], sequence[i + 1]);
432             }
433             if (sequence.Length >= 3)
434             {
435                 StepLeft(handler, sequence[sequence.Length - 2],
436                     ↪ sequence[sequence.Length - 1]);
437             }
438         }
439     }
440     return results;

```

```

437     });
438 }
439
440 public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
441 {
442     return Sync.ExecuteReadOperation(() =>
443     {
444         var results = new HashSet<ulong>();
445         if (sequence.Length > 0)
446         {
447             Links.EnsureEachLinkExists(sequence);
448             var firstElement = sequence[0];
449             if (sequence.Length == 1)
450             {
451                 results.Add(firstElement);
452                 return results;
453             }
454             if (sequence.Length == 2)
455             {
456                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
457                 if (doublet != _constants.Null)
458                 {
459                     results.Add(doublet);
460                 }
461                 return results;
462             }
463             var matcher = new Matcher(this, sequence, results, null);
464             if (sequence.Length >= 2)
465             {
466                 StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
467             }
468             var last = sequence.Length - 2;
469             for (var i = 1; i < last; i++)
470             {
471                 PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
472                     ↪ sequence[i + 1]);
473             }
474             if (sequence.Length >= 3)
475             {
476                 StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
477                     ↪ sequence[sequence.Length - 1]);
478             }
479             return results;
480         }
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             //}
516         }
517     }
518     sb.Append('}');
519 }

```

```

506         //     elementToString(sb, element);
507         //     sb.Append('}');
508         ///}
509         //else
510         elementToString(sb, element);
511         if (sb.Length < MaxSequenceFormatSize)
512         {
513             return true;
514         }
515         sb.Append(insertComma ? ", ..." : "...");
516         return false;
517     });
518 }
519 sb.Append('}');
520 return sb.ToString();
521 }
522
523 public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
    ↪ knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
    ↪ knownElements);
524
525 public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
    ↪ LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
    ↪ Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
    ↪ sequenceLink, elementToString, insertComma, knownElements));
526
527 private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
    ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
    ↪ LinkIndex[] knownElements)
528 {
529     var linksInSequence = new HashSet<ulong>(knownElements);
530     var entered = new HashSet<ulong>();
531     var sb = new StringBuilder();
532     sb.Append('{');
533     if (links.Exists(sequenceLink))
534     {
535         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
536             x => linksInSequence.Contains(x) || links.IsFullPoint(x),
537             ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
538             {
539                 if (insertComma && sb.Length > 1)
540                 {
541                     sb.Append(',');
542                 }
543                 if (entered.Contains(element))
544                 {
545                     sb.Append('{');
546                     elementToString(sb, element);
547                     sb.Append('}');
548                 }
549                 else
550                 {
551                     elementToString(sb, element);
552                 }
553                 if (sb.Length < MaxSequenceFormatSize)
554                 {
555                     return true;
556                 }
557                 sb.Append(insertComma ? ", ..." : "...");
558                 return false;
559             });
560     }
561     sb.Append('}');
562     return sb.ToString();
563 }
564
565 public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
566 {
567     return Sync.ExecuteReadOperation(() =>
568     {
569         if (sequence.Length > 0)
570         {
571             Links.EnsureEachLinkExists(sequence);
572             var results = new HashSet<ulong>();
573             for (var i = 0; i < sequence.Length; i++)
574             {
575                 AllUsagesCore(sequence[i], results);
576             }
577         }
578     });
579 }

```

```

576     var filteredResults = new List<ulong>();
577     var linksInSequence = new HashSet<ulong>(sequence);
578     foreach (var result in results)
579     {
580         var filterPosition = -1;
581         StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
582             ↪ Links.Unsync.GetTarget,
583             ↪ x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
584             ↪ x =>
585             {
586                 if (filterPosition == (sequence.Length - 1))
587                 {
588                     return false;
589                 }
590                 if (filterPosition >= 0)
591                 {
592                     if (x == sequence[filterPosition + 1])
593                     {
594                         filterPosition++;
595                     }
596                     else
597                     {
598                         return false;
599                     }
600                 }
601                 if (filterPosition < 0)
602                 {
603                     if (x == sequence[0])
604                     {
605                         filterPosition = 0;
606                     }
607                     return true;
608                 }
609                 if (filterPosition == (sequence.Length - 1))
610                 {
611                     filteredResults.Add(result);
612                 }
613             }
614         return filteredResults;
615     }
616     return new List<ulong>();
617 }
618
619 public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
620 {
621     return Sync.ExecuteReadOperation(() =>
622     {
623         if (sequence.Length > 0)
624         {
625             Links.EnsureEachLinkExists(sequence);
626             var results = new HashSet<ulong>();
627             for (var i = 0; i < sequence.Length; i++)
628             {
629                 AllUsagesCore(sequence[i], results);
630             }
631             var filteredResults = new HashSet<ulong>();
632             var matcher = new Matcher(this, sequence, filteredResults, null);
633             matcher.AddAllPartialMatchedToResults(results);
634             return filteredResults;
635         }
636         return new HashSet<ulong>();
637     });
638 }
639
640 public bool GetAllPartiallyMatchingSequences2(Func<ulong, bool> handler, params ulong[]
641     ↪ sequence)
642 {
643     return Sync.ExecuteReadOperation(() =>
644     {
645         if (sequence.Length > 0)
646         {
647             Links.EnsureEachLinkExists(sequence);
648
649             var results = new HashSet<ulong>();
650             var filteredResults = new HashSet<ulong>();
651             var matcher = new Matcher(this, sequence, filteredResults, handler);
652             for (var i = 0; i < sequence.Length; i++)

```

```

652         {
653             if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
654             {
655                 return false;
656             }
657         }
658         return true;
659     }
660     return true;
661 });
662 }
663
664 //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
665 //{
666 //    return Sync.ExecuteReadOperation(() =>
667 //    {
668 //        if (sequence.Length > 0)
669 //        {
670 //            _links.EnsureEachLinkIsAnyOrExists(sequence);
671 //
672 //            var firstResults = new HashSet<ulong>();
673 //            var lastResults = new HashSet<ulong>();
674 //
675 //            var first = sequence.First(x => x != LinksConstants.Any);
676 //            var last = sequence.Last(x => x != LinksConstants.Any);
677 //
678 //            AllUsagesCore(first, firstResults);
679 //            AllUsagesCore(last, lastResults);
680 //
681 //            firstResults.IntersectWith(lastResults);
682 //
683 //            //for (var i = 0; i < sequence.Length; i++)
684 //            //    AllUsagesCore(sequence[i], results);
685 //
686 //            var filteredResults = new HashSet<ulong>();
687 //            var matcher = new Matcher(this, sequence, filteredResults, null);
688 //            matcher.AddAllPartialMatchedToResults(firstResults);
689 //            return filteredResults;
690 //        }
691 //
692 //        return new HashSet<ulong>();
693 //    });
694 //}
695
696 public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
697 {
698     return Sync.ExecuteReadOperation(() =>
699     {
700         if (sequence.Length > 0)
701         {
702             Links.EnsureEachLinkIsAnyOrExists(sequence);
703             var firstResults = new HashSet<ulong>();
704             var lastResults = new HashSet<ulong>();
705             var first = sequence.First(x => x != _constants.Any);
706             var last = sequence.Last(x => x != _constants.Any);
707             AllUsagesCore(first, firstResults);
708             AllUsagesCore(last, lastResults);
709             firstResults.IntersectWith(lastResults);
710             //for (var i = 0; i < sequence.Length; i++)
711             //    AllUsagesCore(sequence[i], results);
712             var filteredResults = new HashSet<ulong>();
713             var matcher = new Matcher(this, sequence, filteredResults, null);
714             matcher.AddAllPartialMatchedToResults(firstResults);
715             return filteredResults;
716         }
717         return new HashSet<ulong>();
718     });
719 }
720
721 public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
722     ↪ IList<ulong> sequence)
723 {
724     return Sync.ExecuteReadOperation(() =>
725     {
726         if (sequence.Count > 0)
727         {
728             Links.EnsureEachLinkExists(sequence);
729             var results = new HashSet<LinkIndex>();
730             //var nextResults = new HashSet<ulong>();

```

```

730         //for (var i = 0; i < sequence.Length; i++)
731         //{
732             //    AllUsagesCore(sequence[i], nextResults);
733             //    if (results.IsNullOrEmpty())
734             //    {
735                 //        results = nextResults;
736                 //        nextResults = new HashSet<ulong>();
737             //    }
738             //    else
739             //    {
740                 //        results.IntersectWith(nextResults);
741                 //        nextResults.Clear();
742             //    }
743         //}
744         var collector1 = new AllUsagesCollector1(Links.Unsync, results);
745         collector1.Collect(Links.Unsync.GetLink(sequence[0]));
746         var next = new HashSet<ulong>();
747         for (var i = 1; i < sequence.Count; i++)
748         {
749             var collector = new AllUsagesCollector1(Links.Unsync, next);
750             collector.Collect(Links.Unsync.GetLink(sequence[i]));
751
752             results.IntersectWith(next);
753             next.Clear();
754         }
755         var filteredResults = new HashSet<ulong>();
756         var matcher = new Matcher(this, sequence, filteredResults, null,
757             ↪ readAsElements);
758         matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
759             ↪ x)); // OrderBy is a Hack
760         return filteredResults;
761     }
762     return new HashSet<ulong>();
763 });
764 }
765
766 // Does not work
767 public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
768     ↪ params ulong[] sequence)
769 {
770     var visited = new HashSet<ulong>();
771     var results = new HashSet<ulong>();
772     var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
773         ↪ true; }, readAsElements);
774     var last = sequence.Length - 1;
775     for (var i = 0; i < last; i++)
776     {
777         PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
778     }
779     return results;
780 }
781
782 public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
783 {
784     return Sync.ExecuteReadOperation(() =>
785     {
786         if (sequence.Length > 0)
787         {
788             Links.EnsureEachLinkExists(sequence);
789             //var firstElement = sequence[0];
790             //if (sequence.Length == 1)
791             //{
792                 //    //results.Add(firstElement);
793                 //    return results;
794             //}
795             //if (sequence.Length == 2)
796             //{
797                 //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
798                 //    //if (doublet != Doublets.Links.Null)
799                 //    //    results.Add(doublet);
800                 //    return results;
801             //}
802             //var lastElement = sequence[sequence.Length - 1];
803             //Func<ulong, bool> handler = x =>
804             //{
805                 //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
806                 //        ↪ results.Add(x);
807                 //    ↪ return true;

```

```

803     //});
804     //if (sequence.Length >= 2)
805     //    StepRight(handler, sequence[0], sequence[1]);
806     //var last = sequence.Length - 2;
807     //for (var i = 1; i < last; i++)
808     //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
809     //if (sequence.Length >= 3)
810     //    StepLeft(handler, sequence[sequence.Length - 2],
811     //        ↪ sequence[sequence.Length - 1]);
812     //if (sequence.Length == 1)
813     //    throw new NotImplementedException(); // all sequences, containing
814     //    ↪ this element?
815     //if (sequence.Length == 2)
816     //{
817     //    var results = new List<ulong>();
818     //    PartialStepRight(results.Add, sequence[0], sequence[1]);
819     //    return results;
820     //}
821     //var matches = new List<List<ulong>>();
822     //var last = sequence.Length - 1;
823     //for (var i = 0; i < last; i++)
824     //{
825     //    var results = new List<ulong>();
826     //    //StepRight(results.Add, sequence[i], sequence[i + 1]);
827     //    PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
828     //    if (results.Count > 0)
829     //        matches.Add(results);
830     //    else
831     //        return results;
832     //    if (matches.Count == 2)
833     //    {
834     //        var merged = new List<ulong>();
835     //        for (var j = 0; j < matches[0].Count; j++)
836     //            for (var k = 0; k < matches[1].Count; k++)
837     //                CloseInnerConnections(merged.Add, matches[0][j],
838     //                    ↪ matches[1][k]);
839     //        if (merged.Count > 0)
840     //            matches = new List<List<ulong>> { merged };
841     //        else
842     //            return new List<ulong>();
843     //    }
844     //}
845     //if (matches.Count > 0)
846     //{
847     //    var usages = new HashSet<ulong>();
848     //    for (int i = 0; i < sequence.Length; i++)
849     //    {
850     //        AllUsagesCore(sequence[i], usages);
851     //    }
852     //    //for (int i = 0; i < matches[0].Count; i++)
853     //    //    AllUsagesCore(matches[0][i], usages);
854     //    //usages.UnionWith(matches[0]);
855     //    return usages.ToList();
856     //}
857     var firstLinkUsages = new HashSet<ulong>();
858     AllUsagesCore(sequence[0], firstLinkUsages);
859     firstLinkUsages.Add(sequence[0]);
860     //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
861     //    ↪ sequence[0] }; // or all sequences, containing this element?
862     //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
863     //    ↪ 1).ToList();
864     var results = new HashSet<ulong>();
865     foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
866     ↪ firstLinkUsages, 1))
867     {
868         AllUsagesCore(match, results);
869     }
870     return results.ToList();
871 }
872 return new List<ulong>();
873 }
874 }
875
876 /// <remarks>
877 /// TODO: Может потребоваться ограничение на уровень глубины рекурсии

```

```

874 /// </remarks>
875 public HashSet<ulong> AllUsages(ulong link)
876 {
877     return Sync.ExecuteReadOperation(() =>
878     {
879         var usages = new HashSet<ulong>();
880         AllUsagesCore(link, usages);
881         return usages;
882     });
883 }
884
885 // При сборе всех использований (последовательностей) можно сохранять обратный путь к
886 // той связи с которой начинался поиск (STTTSSSTT),
887 // причём достаточно одного бита для хранения перехода влево или вправо
888 private void AllUsagesCore(ulong link, HashSet<ulong> usages)
889 {
890     bool handler(ulong doublet)
891     {
892         if (usages.Add(doublet))
893         {
894             AllUsagesCore(doublet, usages);
895         }
896         return true;
897     }
898     Links.Unsync.Each(link, _constants.Any, handler);
899     Links.Unsync.Each(_constants.Any, link, handler);
900 }
901
902 public HashSet<ulong> AllBottomUsages(ulong link)
903 {
904     return Sync.ExecuteReadOperation(() =>
905     {
906         var visits = new HashSet<ulong>();
907         var usages = new HashSet<ulong>();
908         AllBottomUsagesCore(link, visits, usages);
909         return usages;
910     });
911 }
912
913 private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
914     ↳ usages)
915 {
916     bool handler(ulong doublet)
917     {
918         if (visits.Add(doublet))
919         {
920             AllBottomUsagesCore(doublet, visits, usages);
921         }
922         return true;
923     }
924     if (Links.Unsync.Count(_constants.Any, link) == 0)
925     {
926         usages.Add(link);
927     }
928     else
929     {
930         Links.Unsync.Each(link, _constants.Any, handler);
931         Links.Unsync.Each(_constants.Any, link, handler);
932     }
933 }
934
935 public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
936 {
937     if (Options.UseSequenceMarker)
938     {
939         var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
940     ↳ Options.MarkedSequenceMatcher, symbol);
941         return counter.Count();
942     }
943     else
944     {
945         var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
946     ↳ symbol);
947         return counter.Count();
948     }
949 }

```



```

947 private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<ulong, bool>
    ↳ outerHandler)
948 {
949     bool handler(ulong doublet)
950     {
951         if (usages.Add(doublet))
952         {
953             if (!outerHandler(doublet))
954             {
955                 return false;
956             }
957             if (!AllUsagesCore1(doublet, usages, outerHandler))
958             {
959                 return false;
960             }
961         }
962         return true;
963     }
964     return Links.Unsync.Each(link, _constants.Any, handler)
965         && Links.Unsync.Each(_constants.Any, link, handler);
966 }
967
968 public void CalculateAllUsages(ulong[] totals)
969 {
970     var calculator = new AllUsagesCalculator(Links, totals);
971     calculator.Calculate();
972 }
973
974 public void CalculateAllUsages2(ulong[] totals)
975 {
976     var calculator = new AllUsagesCalculator2(Links, totals);
977     calculator.Calculate();
978 }
979
980 private class AllUsagesCalculator
981 {
982     private readonly SynchronizedLinks<ulong> _links;
983     private readonly ulong[] _totals;
984
985     public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
986     {
987         _links = links;
988         _totals = totals;
989     }
990
991     public void Calculate() => _links.Each(_constants.Any, _constants.Any,
    ↳ CalculateCore);
992
993     private bool CalculateCore(ulong link)
994     {
995         if (_totals[link] == 0)
996         {
997             var total = 1UL;
998             _totals[link] = total;
999             var visitedChildren = new HashSet<ulong>();
1000             bool linkCalculator(ulong child)
1001             {
1002                 if (link != child && visitedChildren.Add(child))
1003                 {
1004                     total += _totals[child] == 0 ? 1 : _totals[child];
1005                 }
1006                 return true;
1007             }
1008             _links.Unsync.Each(link, _constants.Any, linkCalculator);
1009             _links.Unsync.Each(_constants.Any, link, linkCalculator);
1010             _totals[link] = total;
1011         }
1012         return true;
1013     }
1014 }
1015
1016 private class AllUsagesCalculator2
1017 {
1018     private readonly SynchronizedLinks<ulong> _links;
1019     private readonly ulong[] _totals;
1020
1021     public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1022     {
1023         _links = links;

```

```

1024         _totals = totals;
1025     }
1026
1027     public void Calculate() => _links.Each(_constants.Any, _constants.Any,
        ↪ CalculateCore);
1028
1029     private bool IsElement(ulong link)
1030     {
1031         // _linksInSequence.Contains(link) ||
1032         return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
        ↪ link;
1033     }
1034
1035     private bool CalculateCore(ulong link)
1036     {
1037         // TODO: Проработать защиту от заикливания
1038         // Основано на SequenceWalker.WalkLeft
1039         Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1040         Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1041         Func<ulong, bool> isElement = IsElement;
1042         void visitLeaf(ulong parent)
1043         {
1044             if (link != parent)
1045             {
1046                 _totals[parent]++;
1047             }
1048         }
1049         void visitNode(ulong parent)
1050         {
1051             if (link != parent)
1052             {
1053                 _totals[parent]++;
1054             }
1055         }
1056         var stack = new Stack();
1057         var element = link;
1058         if (isElement(element))
1059         {
1060             visitLeaf(element);
1061         }
1062         else
1063         {
1064             while (true)
1065             {
1066                 if (isElement(element))
1067                 {
1068                     if (stack.Count == 0)
1069                     {
1070                         break;
1071                     }
1072                     element = stack.Pop();
1073                     var source = getSource(element);
1074                     var target = getTarget(element);
1075                     // Обработка элемента
1076                     if (isElement(target))
1077                     {
1078                         visitLeaf(target);
1079                     }
1080                     if (isElement(source))
1081                     {
1082                         visitLeaf(source);
1083                     }
1084                     element = source;
1085                 }
1086                 else
1087                 {
1088                     stack.Push(element);
1089                     visitNode(element);
1090                     element = getTarget(element);
1091                 }
1092             }
1093         }
1094         _totals[link]++;
1095         return true;
1096     }
1097 }
1098
1099 private class AllUsagesCollector
1100 {

```

```

1101     private readonly ILinks<ulong> _links;
1102     private readonly HashSet<ulong> _usages;
1103
1104     public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
1105     {
1106         _links = links;
1107         _usages = usages;
1108     }
1109
1110     public bool Collect(ulong link)
1111     {
1112         if (_usages.Add(link))
1113         {
1114             _links.Each(link, _constants.Any, Collect);
1115             _links.Each(_constants.Any, link, Collect);
1116         }
1117         return true;
1118     }
1119 }
1120
1121 private class AllUsagesCollector1
1122 {
1123     private readonly ILinks<ulong> _links;
1124     private readonly HashSet<ulong> _usages;
1125     private readonly ulong _continue;
1126
1127     public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
1128     {
1129         _links = links;
1130         _usages = usages;
1131         _continue = _links.Constants.Continue;
1132     }
1133
1134     public ulong Collect(ICollection<ulong> link)
1135     {
1136         var linkIndex = _links.GetIndex(link);
1137         if (_usages.Add(linkIndex))
1138         {
1139             _links.Each(Collect, _constants.Any, linkIndex);
1140         }
1141         return _continue;
1142     }
1143 }
1144
1145 private class AllUsagesCollector2
1146 {
1147     private readonly ILinks<ulong> _links;
1148     private readonly BitString _usages;
1149
1150     public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
1151     {
1152         _links = links;
1153         _usages = usages;
1154     }
1155
1156     public bool Collect(ulong link)
1157     {
1158         if (_usages.Add((long)link))
1159         {
1160             _links.Each(link, _constants.Any, Collect);
1161             _links.Each(_constants.Any, link, Collect);
1162         }
1163         return true;
1164     }
1165 }
1166
1167 private class AllUsagesIntersectingCollector
1168 {
1169     private readonly SynchronizedLinks<ulong> _links;
1170     private readonly HashSet<ulong> _intersectWith;
1171     private readonly HashSet<ulong> _usages;
1172     private readonly HashSet<ulong> _enter;
1173
1174     public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
↵ intersectWith, HashSet<ulong> usages)
1175     {
1176         _links = links;
1177         _intersectWith = intersectWith;
1178         _usages = usages;
1179         _enter = new HashSet<ulong>(); // защита от заикливания
1180     }

```

```

1181 public bool Collect(ulong link)
1182 {
1183     if (_enter.Add(link))
1184     {
1185         if (_intersectWith.Contains(link))
1186         {
1187             _usages.Add(link);
1188         }
1189         _links.Unsync.Each(link, _constants.Any, Collect);
1190         _links.Unsync.Each(_constants.Any, link, Collect);
1191     }
1192     return true;
1193 }
1194 }
1195 }
1196
1197 private void CloseInnerConnections(Action<ulong> handler, ulong left, ulong right)
1198 {
1199     TryStepLeftUp(handler, left, right);
1200     TryStepRightUp(handler, right, left);
1201 }
1202
1203 private void AllCloseConnections(Action<ulong> handler, ulong left, ulong right)
1204 {
1205     // Direct
1206     if (left == right)
1207     {
1208         handler(left);
1209     }
1210     var doublet = Links.Unsync.SearchOrDefault(left, right);
1211     if (doublet != _constants.Null)
1212     {
1213         handler(doublet);
1214     }
1215     // Inner
1216     CloseInnerConnections(handler, left, right);
1217     // Outer
1218     StepLeft(handler, left, right);
1219     StepRight(handler, left, right);
1220     PartialStepRight(handler, left, right);
1221     PartialStepLeft(handler, left, right);
1222 }
1223
1224 private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
1225     ↪ HashSet<ulong> previousMatchings, long startAt)
1226 {
1227     if (startAt >= sequence.Length) // ?
1228     {
1229         return previousMatchings;
1230     }
1231     var secondLinkUsages = new HashSet<ulong>();
1232     AllUsagesCore(sequence[startAt], secondLinkUsages);
1233     secondLinkUsages.Add(sequence[startAt]);
1234     var matchings = new HashSet<ulong>();
1235     //for (var i = 0; i < previousMatchings.Count; i++)
1236     foreach (var secondLinkUsage in secondLinkUsages)
1237     {
1238         foreach (var previousMatching in previousMatchings)
1239         {
1240             //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
1241             ↪ secondLinkUsage);
1242             StepRight(matchings.AddAndReturnVoid, previousMatching, secondLinkUsage);
1243             TryStepRightUp(matchings.AddAndReturnVoid, secondLinkUsage,
1244             ↪ previousMatching);
1245             //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
1246             ↪ sequence[startAt]); // почему-то эта ошибочная запись приводит к
1247             ↪ желаемым результатам.
1248             PartialStepRight(matchings.AddAndReturnVoid, previousMatching,
1249             ↪ secondLinkUsage);
1250         }
1251     }
1252     if (matchings.Count == 0)
1253     {
1254         return matchings;
1255     }
1256     return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
1257 }

```

```

1253 private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
    ↳ links, params ulong[] sequence)
1254 {
1255     if (sequence == null)
1256     {
1257         return;
1258     }
1259     for (var i = 0; i < sequence.Length; i++)
1260     {
1261         if (sequence[i] != _constants.Any && sequence[i] != ZeroOrMany &&
            ↳ !links.Exists(sequence[i]))
1262         {
1263             throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                ↳ $"patternSequence[{i}]");
1264         }
1265     }
1266 }
1267
1268 // Pattern Matching -> Key To Triggers
1269 public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1270 {
1271     return Sync.ExecuteReadOperation(() =>
1272     {
1273         patternSequence = Simplify(patternSequence);
1274         if (patternSequence.Length > 0)
1275         {
1276             EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1277             var uniqueSequenceElements = new HashSet<ulong>();
1278             for (var i = 0; i < patternSequence.Length; i++)
1279             {
1280                 if (patternSequence[i] != _constants.Any && patternSequence[i] !=
                    ↳ ZeroOrMany)
1281                 {
1282                     uniqueSequenceElements.Add(patternSequence[i]);
1283                 }
1284             }
1285             var results = new HashSet<ulong>();
1286             foreach (var uniqueSequenceElement in uniqueSequenceElements)
1287             {
1288                 AllUsagesCore(uniqueSequenceElement, results);
1289             }
1290             var filteredResults = new HashSet<ulong>();
1291             var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1292             matcher.AddAllPatternMatchedToResults(results);
1293             return filteredResults;
1294         }
1295         return new HashSet<ulong>();
1296     });
1297 }
1298
1299 // Найти все возможные связи между указанным списком связей.
1300 // Находит связи между всеми указанными связями в любом порядке.
1301 // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
    ↳ несколько раз в последовательности)
1302 public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1303 {
1304     return Sync.ExecuteReadOperation(() =>
1305     {
1306         var results = new HashSet<ulong>();
1307         if (linksToConnect.Length > 0)
1308         {
1309             Links.EnsureEachLinkExists(linksToConnect);
1310             AllUsagesCore(linksToConnect[0], results);
1311             for (var i = 1; i < linksToConnect.Length; i++)
1312             {
1313                 var next = new HashSet<ulong>();
1314                 AllUsagesCore(linksToConnect[i], next);
1315                 results.IntersectWith(next);
1316             }
1317             return results;
1318         }
1319     });
1320 }
1321
1322 public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
1323 {
1324     return Sync.ExecuteReadOperation(() =>
1325     {

```

```

1326     var results = new HashSet<ulong>();
1327     if (linksToConnect.Length > 0)
1328     {
1329         Links.EnsureEachLinkExists(linksToConnect);
1330         var collector1 = new AllUsagesCollector(Links.Unsync, results);
1331         collector1.Collect(linksToConnect[0]);
1332         var next = new HashSet<ulong>();
1333         for (var i = 1; i < linksToConnect.Length; i++)
1334         {
1335             var collector = new AllUsagesCollector(Links.Unsync, next);
1336             collector.Collect(linksToConnect[i]);
1337             results.IntersectWith(next);
1338             next.Clear();
1339         }
1340     }
1341     return results;
1342 });
1343 }
1344
1345 public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
1346 {
1347     return Sync.ExecuteReadOperation(() =>
1348     {
1349         var results = new HashSet<ulong>();
1350         if (linksToConnect.Length > 0)
1351         {
1352             Links.EnsureEachLinkExists(linksToConnect);
1353             var collector1 = new AllUsagesCollector(Links, results);
1354             collector1.Collect(linksToConnect[0]);
1355             //AllUsagesCore(linksToConnect[0], results);
1356             for (var i = 1; i < linksToConnect.Length; i++)
1357             {
1358                 var next = new HashSet<ulong>();
1359                 var collector = new AllUsagesIntersectingCollector(Links, results, next);
1360                 collector.Collect(linksToConnect[i]);
1361                 //AllUsagesCore(linksToConnect[i], next);
1362                 //results.IntersectWith(next);
1363                 results = next;
1364             }
1365         }
1366         return results;
1367     });
1368 }
1369
1370 public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
1371 {
1372     return Sync.ExecuteReadOperation(() =>
1373     {
1374         var results = new BitString((long)Links.Unsync.Count() + 1); // new
1375         ↪ BitArray((int)_links.Total + 1);
1376         if (linksToConnect.Length > 0)
1377         {
1378             Links.EnsureEachLinkExists(linksToConnect);
1379             var collector1 = new AllUsagesCollector2(Links.Unsync, results);
1380             collector1.Collect(linksToConnect[0]);
1381             for (var i = 1; i < linksToConnect.Length; i++)
1382             {
1383                 var next = new BitString((long)Links.Unsync.Count() + 1); //new
1384                 ↪ BitArray((int)_links.Total + 1);
1385                 var collector = new AllUsagesCollector2(Links.Unsync, next);
1386                 collector.Collect(linksToConnect[i]);
1387                 results = results.And(next);
1388             }
1389         }
1390         return results.GetSetUInt64Indices();
1391     });
1392 }
1393
1394 private static ulong[] Simplify(ulong[] sequence)
1395 {
1396     // Считаем новый размер последовательности
1397     long newLength = 0;
1398     var zeroOrManyStepped = false;
1399     for (var i = 0; i < sequence.Length; i++)
1400     {
1401         if (sequence[i] == ZeroOrMany)
1402         {
1403             if (zeroOrManyStepped)

```

```

1402         {
1403             continue;
1404         }
1405         zeroOrManyStepped = true;
1406     }
1407     else
1408     {
1409         //if (zeroOrManyStepped) Is it efficient?
1410         zeroOrManyStepped = false;
1411     }
1412     newLength++;
1413 }
1414 // Строим новую последовательность
1415 zeroOrManyStepped = false;
1416 var newSequence = new ulong[newLength];
1417 long j = 0;
1418 for (var i = 0; i < sequence.Length; i++)
1419 {
1420     //var current = zeroOrManyStepped;
1421     //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
1422     //if (current && zeroOrManyStepped)
1423     //    continue;
1424     //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1425     //if (zeroOrManyStepped && newZeroOrManyStepped)
1426     //    continue;
1427     //zeroOrManyStepped = newZeroOrManyStepped;
1428     if (sequence[i] == ZeroOrMany)
1429     {
1430         if (zeroOrManyStepped)
1431         {
1432             continue;
1433         }
1434         zeroOrManyStepped = true;
1435     }
1436     else
1437     {
1438         //if (zeroOrManyStepped) Is it efficient?
1439         zeroOrManyStepped = false;
1440     }
1441     newSequence[j++] = sequence[i];
1442 }
1443 return newSequence;
1444 }
1445
1446 public static void TestSimplify()
1447 {
1448     var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany,
1449     ↪ ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
1450     var simplifiedSequence = Simplify(sequence);
1451 }
1452
1453 public List<ulong> GetSimilarSequences() => new List<ulong>();
1454
1455 public void Prediction()
1456 {
1457     //_links
1458     //sequences
1459 }
1460
1461 #region From Triplets
1462
1463 //public static void DeleteSequence(Link sequence)
1464 //{
1465 //}
1466
1467 public List<ulong> CollectMatchingSequences(ulong[] links)
1468 {
1469     if (links.Length == 1)
1470     {
1471         throw new Exception("Подпоследовательности с одним элементом не
1472         ↪ поддерживаются.");
1473     }
1474     var leftBound = 0;
1475     var rightBound = links.Length - 1;
1476     var left = links[leftBound++];
1477     var right = links[rightBound--];
1478     var results = new List<ulong>();
1479     CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
1480     return results;

```

```

1479     }
1480
1481     private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
1482     ↪ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1483     {
1484         var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1485         var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1486         if (leftLinkTotalReferers <= rightLinkTotalReferers)
1487         {
1488             var nextLeftLink = middleLinks[leftBound];
1489             var elements = GetRightElements(leftLink, nextLeftLink);
1490             if (leftBound <= rightBound)
1491             {
1492                 for (var i = elements.Length - 1; i >= 0; i--)
1493                 {
1494                     var element = elements[i];
1495                     if (element != 0)
1496                     {
1497                         CollectMatchingSequences(element, leftBound + 1, middleLinks,
1498                         ↪ rightLink, rightBound, ref results);
1499                     }
1500                 }
1501             }
1502             else
1503             {
1504                 for (var i = elements.Length - 1; i >= 0; i--)
1505                 {
1506                     var element = elements[i];
1507                     if (element != 0)
1508                     {
1509                         results.Add(element);
1510                     }
1511                 }
1512             }
1513         }
1514         else
1515         {
1516             var nextRightLink = middleLinks[rightBound];
1517             var elements = GetLeftElements(rightLink, nextRightLink);
1518             if (leftBound <= rightBound)
1519             {
1520                 for (var i = elements.Length - 1; i >= 0; i--)
1521                 {
1522                     var element = elements[i];
1523                     if (element != 0)
1524                     {
1525                         CollectMatchingSequences(leftLink, leftBound, middleLinks,
1526                         ↪ elements[i], rightBound - 1, ref results);
1527                     }
1528                 }
1529             }
1530             else
1531             {
1532                 for (var i = elements.Length - 1; i >= 0; i--)
1533                 {
1534                     var element = elements[i];
1535                     if (element != 0)
1536                     {
1537                         results.Add(element);
1538                     }
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         }
1558     }

```



```

1554         return true;
1555     });
1556     if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
1557     {
1558         result[4] = startLink;
1559     }
1560     return result;
1561 }
1562
1563 public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
1564 {
1565     var added = 0;
1566     Links.Each(startLink, _constants.Any, couple =>
1567     {
1568         if (couple != startLink)
1569         {
1570             var coupleTarget = Links.GetTarget(couple);
1571             if (coupleTarget == rightLink)
1572             {
1573                 result[offset] = couple;
1574                 if (++added == 2)
1575                 {
1576                     return false;
1577                 }
1578             }
1579             else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker
1580                 == Net.And &&
1581             {
1582                 result[offset + 1] = couple;
1583                 if (++added == 2)
1584                 {
1585                     return false;
1586                 }
1587             }
1588             return true;
1589         });
1590     return added > 0;
1591 }
1592
1593 public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
1594 {
1595     var result = new ulong[5];
1596     TryStepLeft(startLink, leftLink, result, 0);
1597     Links.Each(startLink, _constants.Any, couple =>
1598     {
1599         if (couple != startLink)
1600         {
1601             if (TryStepLeft(couple, leftLink, result, 2))
1602             {
1603                 return false;
1604             }
1605         }
1606         return true;
1607     });
1608     if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
1609     {
1610         result[4] = leftLink;
1611     }
1612     return result;
1613 }
1614
1615 public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
1616 {
1617     var added = 0;
1618     Links.Each(_constants.Any, startLink, couple =>
1619     {
1620         if (couple != startLink)
1621         {
1622             var coupleSource = Links.GetSource(couple);
1623             if (coupleSource == leftLink)
1624             {
1625                 result[offset] = couple;
1626                 if (++added == 2)
1627                 {
1628                     return false;
1629                 }
1630             }

```

```

1631         else if (Links.GetTarget(coupleSource) == leftLink) // coupleSource.Linker
1632             ↪ == Net.And &&
1633         {
1634             result[offset + 1] = couple;
1635             if (++added == 2)
1636             {
1637                 return false;
1638             }
1639         }
1640     return true;
1641 });
1642 return added > 0;
1643 }
1644
1645 #endregion
1646
1647 #region Walkers
1648
1649 public class PatternMatcher : RightSequenceWalker<ulong>
1650 {
1651     private readonly Sequences _sequences;
1652     private readonly ulong[] _patternSequence;
1653     private readonly HashSet<LinkIndex> _linksInSequence;
1654     private readonly HashSet<LinkIndex> _results;
1655
1656     #region Pattern Match
1657
1658     enum PatternBlockType
1659     {
1660         Undefined,
1661         Gap,
1662         Elements
1663     }
1664
1665     struct PatternBlock
1666     {
1667         public PatternBlockType Type;
1668         public long Start;
1669         public long Stop;
1670     }
1671
1672     private readonly List<PatternBlock> _pattern;
1673     private int _patternPosition;
1674     private long _sequencePosition;
1675
1676     #endregion
1677
1678     public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
1679         ↪ HashSet<LinkIndex> results)
1680         : base(sequences.Links.Unsync, new DefaultStack<ulong>())
1681     {
1682         _sequences = sequences;
1683         _patternSequence = patternSequence;
1684         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
1685             ↪ _constants.Any && x != ZeroOrMany));
1686         _results = results;
1687         _pattern = CreateDetailedPattern();
1688     }
1689
1690     protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
1691         ↪ base.IsElement(link);
1692
1693     public bool PatternMatch(LinkIndex sequenceToMatch)
1694     {
1695         _patternPosition = 0;
1696         _sequencePosition = 0;
1697         foreach (var part in Walk(sequenceToMatch))
1698         {
1699             if (!PatternMatchCore(part))
1700             {
1701                 break;
1702             }
1703         }
1704         return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
1705             ↪ - 1 && _pattern[_patternPosition].Start == 0);
1706     }
1707
1708     private List<PatternBlock> CreateDetailedPattern()
1709     {
1710         var pattern = new List<PatternBlock>();
1711     }

```

```

1707 var patternBlock = new PatternBlock();
1708 for (var i = 0; i < _patternSequence.Length; i++)
1709 {
1710     if (patternBlock.Type == PatternBlockType.Undefined)
1711     {
1712         if (_patternSequence[i] == _constants.Any)
1713         {
1714             patternBlock.Type = PatternBlockType.Gap;
1715             patternBlock.Start = 1;
1716             patternBlock.Stop = 1;
1717         }
1718         else if (_patternSequence[i] == ZeroOrMany)
1719         {
1720             patternBlock.Type = PatternBlockType.Gap;
1721             patternBlock.Start = 0;
1722             patternBlock.Stop = long.MaxValue;
1723         }
1724         else
1725         {
1726             patternBlock.Type = PatternBlockType.Elements;
1727             patternBlock.Start = i;
1728             patternBlock.Stop = i;
1729         }
1730     }
1731     else if (patternBlock.Type == PatternBlockType.Elements)
1732     {
1733         if (_patternSequence[i] == _constants.Any)
1734         {
1735             pattern.Add(patternBlock);
1736             patternBlock = new PatternBlock
1737             {
1738                 Type = PatternBlockType.Gap,
1739                 Start = 1,
1740                 Stop = 1
1741             };
1742         }
1743         else if (_patternSequence[i] == ZeroOrMany)
1744         {
1745             pattern.Add(patternBlock);
1746             patternBlock = new PatternBlock
1747             {
1748                 Type = PatternBlockType.Gap,
1749                 Start = 0,
1750                 Stop = long.MaxValue
1751             };
1752         }
1753         else
1754         {
1755             patternBlock.Stop = i;
1756         }
1757     }
1758     else // patternBlock.Type == PatternBlockType.Gap
1759     {
1760         if (_patternSequence[i] == _constants.Any)
1761         {
1762             patternBlock.Start++;
1763             if (patternBlock.Stop < patternBlock.Start)
1764             {
1765                 patternBlock.Stop = patternBlock.Start;
1766             }
1767         }
1768         else if (_patternSequence[i] == ZeroOrMany)
1769         {
1770             patternBlock.Stop = long.MaxValue;
1771         }
1772         else
1773         {
1774             pattern.Add(patternBlock);
1775             patternBlock = new PatternBlock
1776             {
1777                 Type = PatternBlockType.Elements,
1778                 Start = i,
1779                 Stop = i
1780             };
1781         }
1782     }
1783 }
1784 if (patternBlock.Type != PatternBlockType.Undefined)
1785 {
1786     pattern.Add(patternBlock);

```

```

1787     }
1788     return pattern;
1789 }
1790
1791 // match: search for regexp anywhere in text
1792 //int match(char* regexp, char* text)
1793 //{
1794 //    do
1795 //    {
1796 //        } while (*text++ != '\0');
1797 //    return 0;
1798 //}
1799
1800 // matchhere: search for regexp at beginning of text
1801 //int matchhere(char* regexp, char* text)
1802 //{
1803 //    if (regexp[0] == '\0')
1804 //        return 1;
1805 //    if (regexp[1] == '*')
1806 //        return matchstar(regexp[0], regexp + 2, text);
1807 //    if (regexp[0] == '$' && regexp[1] == '\0')
1808 //        return *text == '\0';
1809 //    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
1810 //        return matchhere(regexp + 1, text + 1);
1811 //    return 0;
1812 //}
1813
1814 // matchstar: search for c*regexp at beginning of text
1815 //int matchstar(int c, char* regexp, char* text)
1816 //{
1817 //    do
1818 //    {
1819 //        /* a * matches zero or more instances */
1820 //        if (matchhere(regexp, text))
1821 //            return 1;
1822 //    } while (*text != '\0' && (*text++ == c || c == '.'));
1823 //    return 0;
1824 //}
1825
1826 //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out
1827 ↪ long maximumGap)
1828 //{
1829 //    mininumGap = 0;
1830 //    maximumGap = 0;
1831 //    element = 0;
1832 //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
1833 //    {
1834 //        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
1835 //            mininumGap++;
1836 //        else if (_patternSequence[_patternPosition] == ZeroOrMany)
1837 //            maximumGap = long.MaxValue;
1838 //        else
1839 //            break;
1840 //    }
1841 //    if (maximumGap < mininumGap)
1842 //        maximumGap = mininumGap;
1843 //}
1844
1845 private bool PatternMatchCore(LinkIndex element)
1846 {
1847     if (_patternPosition >= _pattern.Count)
1848     {
1849         _patternPosition = -2;
1850         return false;
1851     }
1852     var currentPatternBlock = _pattern[_patternPosition];
1853     if (currentPatternBlock.Type == PatternBlockType.Gap)
1854     {
1855         //var currentMatchingBlockLength = (_sequencePosition -
1856 ↪ _lastMatchedBlockPosition);
1857         if (_sequencePosition < currentPatternBlock.Start)
1858         {
1859             _sequencePosition++;
1860             return true; // Двигаемся дальше
1861         }
1862         // Это последний блок
1863         if (_pattern.Count == _patternPosition + 1)
1864         {

```

```

1863         _patternPosition++;
1864         _sequencePosition = 0;
1865         return false; // Полное соответствие
1866     }
1867     else
1868     {
1869         if (_sequencePosition > currentPatternBlock.Stop)
1870         {
1871             return false; // Соответствие невозможно
1872         }
1873         var nextPatternBlock = _pattern[_patternPosition + 1];
1874         if (_patternSequence[nextPatternBlock.Start] == element)
1875         {
1876             if (nextPatternBlock.Start < nextPatternBlock.Stop)
1877             {
1878                 _patternPosition++;
1879                 _sequencePosition = 1;
1880             }
1881             else
1882             {
1883                 _patternPosition += 2;
1884                 _sequencePosition = 0;
1885             }
1886         }
1887     }
1888 }
1889 else // currentPatternBlock.Type == PatternBlockType.Elements
1890 {
1891     var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1892     if (_patternSequence[patternElementPosition] != element)
1893     {
1894         return false; // Соответствие невозможно
1895     }
1896     if (patternElementPosition == currentPatternBlock.Stop)
1897     {
1898         _patternPosition++;
1899         _sequencePosition = 0;
1900     }
1901     else
1902     {
1903         _sequencePosition++;
1904     }
1905 }
1906 return true;
1907 //if (_patternSequence[_patternPosition] != element)
1908 //    return false;
1909 //else
1910 //{
1911 //    _sequencePosition++;
1912 //    _patternPosition++;
1913 //    return true;
1914 //}
1915 ///////
1916 //if (_filterPosition == _patternSequence.Length)
1917 //{
1918 //    _filterPosition = -2; // Длиннее чем нужно
1919 //    return false;
1920 //}
1921 //if (element != _patternSequence[_filterPosition])
1922 //{
1923 //    _filterPosition = -1;
1924 //    return false; // Начинается иначе
1925 //}
1926 //_filterPosition++;
1927 //if (_filterPosition == (_patternSequence.Length - 1))
1928 //    return false;
1929 //if (_filterPosition >= 0)
1930 //{
1931 //    if (element == _patternSequence[_filterPosition + 1])
1932 //        _filterPosition++;
1933 //    else
1934 //        return false;
1935 //}
1936 //if (_filterPosition < 0)
1937 //{
1938 //    if (element == _patternSequence[0])
1939 //        _filterPosition = 0;
1940 //}
1941 }

```

```

1942
1943         public void AddAllPatternMatchedToResults(IEnumerable<ulong> sequencesToMatch)
1944         {
1945             foreach (var sequenceToMatch in sequencesToMatch)
1946             {
1947                 if (PatternMatch(sequenceToMatch))
1948                 {
1949                     _results.Add(sequenceToMatch);
1950                 }
1951             }
1952         }
1953     }
1954
1955     #endregion
1956 }
1957 }

```

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

```

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

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4  using Platform.Collections.Stacks;
5  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
6  using Platform.Data.Doublets.Sequences.Frequencies.Counters;
7  using Platform.Data.Doublets.Sequences.Converters;
8  using Platform.Data.Doublets.Sequences.CriteriaMatchers;
9  using Platform.Data.Doublets.Sequences.Walkers;
10 using Platform.Data.Doublets.Sequences.Indexes;
11
12 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
13
14 namespace Platform.Data.Doublets.Sequences
15 {
16     public class SequencesOptions<TLink> // TODO: To use type parameter <TLink> the
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     }
30 }

```

```

27 public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting { get; set; }
28 public bool EnforceSingleSequenceVersionOnWriteBasedOnNew { get; set; }
29
30 public MarkedSequenceCriterionMatcher<TLink> MarkedSequenceMatcher { get; set; }
31 public IConverter<IList<TLink>, TLink> LinksToSequenceConverter { get; set; }
32 public ISequenceIndex<TLink> Index { get; set; }
33 public ISequenceWalker<TLink> Walker { get; set; }
34
35 // TODO: Реализовать компактификацию при чтении
36 //public bool EnforceSingleSequenceVersionOnRead { get; set; }
37 //public bool UseRequestMarker { get; set; }
38 //public bool StoreRequestResults { get; set; }
39
40 public void InitOptions(ISynchronizedLinks<TLink> links)
41 {
42     if (UseSequenceMarker)
43     {
44         if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
45         {
46             SequenceMarkerLink = links.CreatePoint();
47         }
48         else
49         {
50             if (!links.Exists(SequenceMarkerLink))
51             {
52                 var link = links.CreatePoint();
53                 if (!_equalityComparer.Equals(link, SequenceMarkerLink))
54                 {
55                     throw new InvalidOperationException("Cannot recreate sequence marker
56                                     ↪ link.");
57                 }
58             }
59             if (MarkedSequenceMatcher == null)
60             {
61                 MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
62                                     ↪ SequenceMarkerLink);
63             }
64             var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
65             if (UseCompression)
66             {
67                 if (LinksToSequenceConverter == null)
68                 {
69                     ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
70                     if (UseSequenceMarker)
71                     {
72                         totalSequenceSymbolFrequencyCounter = new
73                             ↪ TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
74                             ↪ MarkedSequenceMatcher);
75                     }
76                     else
77                     {
78                         totalSequenceSymbolFrequencyCounter = new
79                             ↪ TotalSequenceSymbolFrequencyCounter<TLink>(links);
80                     }
81                     var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
82                                     ↪ totalSequenceSymbolFrequencyCounter);
83                     var compressingConverter = new CompressingConverter<TLink>(links,
84                                     ↪ balancedVariantConverter, doubletFrequenciesCache);
85                     LinksToSequenceConverter = compressingConverter;
86                 }
87             }
88             else
89             {
90                 if (LinksToSequenceConverter == null)
91                 {
92                     LinksToSequenceConverter = balancedVariantConverter;
93                 }
94             }
95             if (UseIndex && Index == null)
96             {
97                 Index = new SequenceIndex<TLink>(links);
98             }
99             if (Walker == null)
100             {
101                 Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
102             }
103         }
104     }
105 }

```

```

98     }
99
100     public void ValidateOptions()
101     {
102         if (UseGarbageCollection && !UseSequenceMarker)
103         {
104             throw new NotSupportedException("To use garbage collection UseSequenceMarker
105                 ↪ option must be on.");
106         }
107     }
108 }

```

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

```

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

```

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

```

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

```

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

```

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

```



```

10 #endif
11
12 namespace Platform.Data.Doublets.Sequences.Walkers
13 {
14     public class LeveledSequenceWalker<TLink> : LinksOperatorBase<TLink>, ISequenceWalker<TLink>
15     {
16         private static readonly EqualityComparer<TLink> _equalityComparer =
17             ↪ EqualityComparer<TLink>.Default;
18
19         private readonly Func<TLink, bool> _isElement;
20
21         public LeveledSequenceWalker(ILinks<TLink> links, Func<TLink, bool> isElement) :
22             ↪ base(links) => _isElement = isElement;
23
24         public LeveledSequenceWalker(ILinks<TLink> links) : base(links) => _isElement =
25             ↪ Links.IsPartialPoint;
26
27         public IEnumerable<TLink> Walk(TLink sequence) => ToArray(sequence);
28
29         public TLink[] ToArray(TLink sequence)
30         {
31             var length = 1;
32             var array = new TLink[length];
33             array[0] = sequence;
34             if (_isElement(sequence))
35             {
36                 return array;
37             }
38             bool hasElements;
39             do
40             {
41                 length *= 2;
42                 #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                 #if USEARRAYPOOL
74                     if (array.Length > 1)
75                     {
76                         ArrayPool.Free(array);
77                     }
78                 #endif
79                 array = nextArray;
80             }
81             while (hasElements);
82             var filledElementsCount = CountFilledElements(array);
83             if (filledElementsCount == array.Length)
84             {
85                 return array;
86             }
87             else
88             {
89                 return CopyFilledElements(array, filledElementsCount);
90             }
91         }
92     }
93 }

```

```

87     }
88 }
89
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 private static TLink[] CopyFilledElements(TLink[] array, int filledElementsCount)
92 {
93     var finalArray = new TLink[filledElementsCount];
94     for (int i = 0, j = 0; i < array.Length; i++)
95     {
96         if (!_equalityComparer.Equals(array[i], default))
97         {
98             finalArray[j] = array[i];
99             j++;
100         }
101     }
102     #if USEARRAYPOOL
103         ArrayPool.Free(array);
104     #endif
105     return finalArray;
106 }
107
108 [MethodImpl(MethodImplOptions.AggressiveInlining)]
109 private static int CountFilledElements(TLink[] array)
110 {
111     var count = 0;
112     for (var i = 0; i < array.Length; i++)
113     {
114         if (!_equalityComparer.Equals(array[i], default))
115         {
116             count++;
117         }
118     }
119     return count;
120 }
121 }
122 }

```

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

```

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

```

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

```
1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Collections.Stacks;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Sequences.Walkers
9 {
10     public abstract class SequenceWalkerBase<TLink> : LinksOperatorBase<TLink>,
11         ↳ ISequenceWalker<TLink>
12     {
13         private readonly IStack<TLink> _stack;
14         private readonly Func<TLink, bool> _isElement;
15
16         protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
17             ↳ isElement) : base(links)
18         {
19             _stack = stack;
20             _isElement = isElement;
21         }
22
23         protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : this(links,
24             ↳ stack, links.IsPartialPoint)
25         {
26         }
27
28         public IEnumerable<TLink> Walk(TLink sequence)
29         {
30             _stack.Clear();
31             var element = sequence;
32             if (IsElement(element))
33             {
34                 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             [MethodImpl(MethodImplOptions.AggressiveInlining)]
62             protected virtual bool IsElement(TLink elementLink) => _isElement(elementLink);
63
64             [MethodImpl(MethodImplOptions.AggressiveInlining)]
65             protected abstract TLink GetNextElementAfterPop(TLink element);
66
67             [MethodImpl(MethodImplOptions.AggressiveInlining)]
68             protected abstract TLink GetNextElementAfterPush(TLink element);
69
70             [MethodImpl(MethodImplOptions.AggressiveInlining)]
71             protected abstract IEnumerable<TLink> WalkContents(TLink element);
72         }
73     }
74 }
```

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

```
1 using System.Collections.Generic;
2 using Platform.Collections.Stacks;
3
```

```

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

```

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

```

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

```

./Platform.Data.Doublets/SynchronizedLinks.cs

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Data.Constants;
4  using Platform.Data.Doublets;
5  using Platform.Threading.Synchronization;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets
10 {
11     /// <remarks>
12     /// TODO: Autogeneration of synchronized wrapper (decorator).
13     /// TODO: Try to unfold code of each method using IL generation for performance improvements.
14     /// TODO: Or even to unfold multiple layers of implementations.
15     /// </remarks>
16     public class SynchronizedLinks<T> : ISynchronizedLinks<T>
17     {
18         public LinksCombinedConstants<T, T, int> Constants { get; }
19         public ISynchronization SyncRoot { get; }
20     }
21 }

```

```

20     public ILinks<T> Sync { get; }
21     public ILinks<T> Unsync { get; }
22
23     public SynchronizedLinks(ILinks<T> links) : this(new ReaderWriterLockSynchronization(),
24         ↳ links) { }
25
26     public SynchronizedLinks(ISynchronization synchronization, ILinks<T> links)
27     {
28         SyncRoot = synchronization;
29         Sync = this;
30         Unsync = links;
31         Constants = links.Constants;
32     }
33
34     public T Count(IList<T> restriction) => SyncRoot.ExecuteReadOperation(restriction,
35         ↳ Unsync.Count);
36     public T Each(Func<IList<T>, T> handler, IList<T> restrictions) =>
37         ↳ SyncRoot.ExecuteReadOperation(handler, restrictions, (handler1, restrictions1) =>
38         ↳ Unsync.Each(handler1, restrictions1));
39     public T Create() => SyncRoot.ExecuteWriteOperation(Unsync.Create);
40     public T Update(IList<T> restrictions) => SyncRoot.ExecuteWriteOperation(restrictions,
41         ↳ Unsync.Update);
42     public void Delete(T link) => SyncRoot.ExecuteWriteOperation(link, Unsync.Delete);
43
44     //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
45     ↳ IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
46     //{
47     //    if (restriction != null && substitution != null &&
48     ↳ !substitution.EqualTo(restriction))
49     //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
50     ↳ substitution, substitutedHandler, Unsync.Trigger);
51     //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
52     ↳ substitutedHandler, Unsync.Trigger);
53     //}
54 }
55 }

```

./Platform.Data.Doublets/UInt64Link.cs

```

1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using Platform.Exceptions;
5  using Platform.Ranges;
6  using Platform.Singletons;
7  using Platform.Collections.Lists;
8  using Platform.Data.Constants;
9
10 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12 namespace Platform.Data.Doublets
13 {
14     /// <summary>
15     /// Структура описывающая уникальную связь.
16     /// </summary>
17     public struct UInt64Link : IEquatable<UInt64Link>, IReadOnlyList<ulong>, IList<ulong>
18     {
19         private static readonly LinksCombinedConstants<bool, ulong, int> _constants =
20             ↳ Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
21
22         private const int Length = 3;
23
24         public readonly ulong Index;
25         public readonly ulong Source;
26         public readonly ulong Target;
27
28         public static readonly UInt64Link Null = new UInt64Link();
29
30         public UInt64Link(params ulong[] values)
31         {
32             Index = values.Length > _constants.IndexPart ? values[_constants.IndexPart] :
33                 ↳ _constants.Null;
34             Source = values.Length > _constants.SourcePart ? values[_constants.SourcePart] :
35                 ↳ _constants.Null;
36             Target = values.Length > _constants.TargetPart ? values[_constants.TargetPart] :
37                 ↳ _constants.Null;
38         }
39
40         public UInt64Link(IList<ulong> values)
41         {

```

```

38         Index = values.Count > _constants.IndexPart ? values[_constants.IndexPart] :
39         ↪ _constants.Null;
40         Source = values.Count > _constants.SourcePart ? values[_constants.SourcePart] :
41         ↪ _constants.Null;
42         Target = values.Count > _constants.TargetPart ? values[_constants.TargetPart] :
43         ↪ _constants.Null;
44     }
45
46     public UInt64Link(ulong index, ulong source, ulong target)
47     {
48         Index = index;
49         Source = source;
50         Target = target;
51     }
52
53     public UInt64Link(ulong source, ulong target)
54     : this(_constants.Null, source, target)
55     {
56         Source = source;
57         Target = target;
58     }
59
60     public static UInt64Link Create(ulong source, ulong target) => new UInt64Link(source,
61     ↪ target);
62
63     public override int GetHashCode() => (Index, Source, Target).GetHashCode();
64
65     public bool IsNull() => Index == _constants.Null
66     && Source == _constants.Null
67     && Target == _constants.Null;
68
69     public override bool Equals(object other) => other is UInt64Link &&
70     ↪ Equals((UInt64Link)other);
71
72     public bool Equals(UInt64Link other) => Index == other.Index
73     && Source == other.Source
74     && Target == other.Target;
75
76     public static string ToString(ulong index, ulong source, ulong target) => $"{index}:
77     ↪ {source}->{target}";
78
79     public static string ToString(ulong source, ulong target) => $"{source}->{target}";
80
81     public static implicit operator ulong[] (UInt64Link link) => link.ToArray();
82
83     public static implicit operator UInt64Link(ulong[] linkArray) => new
84     ↪ UInt64Link(linkArray);
85
86     public override string ToString() => Index == _constants.Null ? ToString(Source, Target)
87     ↪ : ToString(Index, Source, Target);
88
89     #region IList
90
91     public ulong this[int index]
92     {
93         get
94         {
95             Ensure.OnDebug.ArgumentInRange(index, new Range<int>(0, Length - 1),
96             ↪ nameof(index));
97             if (index == _constants.IndexPart)
98             {
99                 return Index;
100             }
101             if (index == _constants.SourcePart)
102             {
103                 return Source;
104             }
105             if (index == _constants.TargetPart)
106             {
107                 return Target;
108             }
109             throw new NotSupportedException(); // Impossible path due to
110             ↪ Ensure.ArgumentInRange
111         }
112         set => throw new NotSupportedException();
113     }
114
115     public int Count => Length;
116
117     public bool IsReadOnly => true;

```

```

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

```

./Platform.Data.Doublets/UInt64LinkExtensions.cs

```

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

```

./Platform.Data.Doublets/UInt64LinksExtensions.cs

```

1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using Platform.Singletons;
5  using Platform.Data.Constants;
6  using Platform.Data.Exceptions;
7  using Platform.Data.Doublets.Unicode;
8

```

```

9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets
12 {
13     public static class UInt64LinksExtensions
14     {
15         public static readonly LinksCombinedConstants<bool, ulong, int> Constants =
16             ↪ Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
17
18         public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
19
20         public static void EnsureEachLinkExists(this ILinks<ulong> links, IList<ulong> sequence)
21         {
22             if (sequence == null)
23             {
24                 return;
25             }
26             for (var i = 0; i < sequence.Count; i++)
27             {
28                 if (!links.Exists(sequence[i]))
29                 {
30                     throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
31                         ↪ $"sequence[{i}]");
32                 }
33             }
34
35         public static void EnsureEachLinkIsAnyOrExists(this ILinks<ulong> links, IList<ulong>
36             ↪ sequence)
37         {
38             if (sequence == null)
39             {
40                 return;
41             }
42             for (var i = 0; i < sequence.Count; i++)
43             {
44                 if (sequence[i] != Constants.Any && !links.Exists(sequence[i]))
45                 {
46                     throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
47                         ↪ $"sequence[{i}]");
48                 }
49             }
50
51         public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
52         {
53             if (sequence == null)
54             {
55                 return false;
56             }
57             var constants = links.Constants;
58             for (var i = 0; i < sequence.Length; i++)
59             {
60                 if (sequence[i] == constants.Any)
61                 {
62                     return true;
63                 }
64             }
65             return false;
66         }
67
68         public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
69             ↪ Func<UInt64Link, bool> isElement, bool renderIndex = false, bool renderDebug = false)
70         {
71             var sb = new StringBuilder();
72             var visited = new HashSet<ulong>();
73             links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
74                 ↪ innerSb.Append(link.Index), renderIndex, renderDebug);
75             return sb.ToString();
76         }
77
78         public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
79             ↪ Func<UInt64Link, bool> isElement, Action<StringBuilder, UInt64Link> appendElement,
80             ↪ bool renderIndex = false, bool renderDebug = false)
81         {
82             var sb = new StringBuilder();
83             var visited = new HashSet<ulong>();

```



```

78         links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
79             ↳ renderDebug);
80         return sb.ToString();
81     }
82     public static void AppendStructure(this ILinks<ulong> links, StringBuilder sb,
83     ↳ HashSet<ulong> visited, ulong linkIndex, Func<UInt64Link, bool> isElement,
84     ↳ Action<StringBuilder, UInt64Link> appendElement, bool renderIndex = false, bool
85     ↳ renderDebug = false)
86     {
87         if (sb == null)
88         {
89             throw new ArgumentNullException(nameof(sb));
90         }
91         if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
92             ↳ Constants.Itself)
93         {
94             return;
95         }
96         if (links.Exists(linkIndex))
97         {
98             if (visited.Add(linkIndex))
99             {
100                 sb.Append('(');
101                 var link = new UInt64Link(links.GetLink(linkIndex));
102                 if (renderIndex)
103                 {
104                     sb.Append(link.Index);
105                     sb.Append(':');
106                 }
107                 if (link.Source == link.Index)
108                 {
109                     sb.Append(link.Index);
110                 }
111                 else
112                 {
113                     var source = new UInt64Link(links.GetLink(link.Source));
114                     if (isElement(source))
115                     {
116                         appendElement(sb, source);
117                     }
118                     else
119                     {
120                         links.AppendStructure(sb, visited, source.Index, isElement,
121                             ↳ appendElement, renderIndex);
122                     }
123                 }
124                 sb.Append(' ');
125                 if (link.Target == link.Index)
126                 {
127                     sb.Append(link.Index);
128                 }
129                 else
130                 {
131                     var target = new UInt64Link(links.GetLink(link.Target));
132                     if (isElement(target))
133                     {
134                         appendElement(sb, target);
135                     }
136                     else
137                     {
138                         links.AppendStructure(sb, visited, target.Index, isElement,
139                             ↳ appendElement, renderIndex);
140                     }
141                 }
142                 sb.Append(')');
143             }
144             else
145             {
146                 if (renderDebug)
147                 {
148                     sb.Append('*');
149                 }
150                 sb.Append(linkIndex);
151             }
152         }
153     }
154     else
155     {

```

```

149         if (renderDebug)
150         {
151             sb.Append('~');
152         }
153         sb.Append(linkIndex);
154     }
155 }
156 }
157 }

```

./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs

```

1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4  using System.IO;
5  using System.Runtime.CompilerServices;
6  using System.Threading;
7  using System.Threading.Tasks;
8  using Platform.Disposables;
9  using Platform.Timestamps;
10 using Platform.Unsafe;
11 using Platform.IO;
12 using Platform.Data.Doublets.Decorators;
13
14 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16 namespace Platform.Data.Doublets
17 {
18     public class UInt64LinksTransactionsLayer : LinksDisposableDecoratorBase //-V3073
19     {
20         /// <remarks>
21         /// Альтернативные варианты хранения трансформации (элемента транзакции):
22         ///
23         /// private enum TransitionType
24         /// {
25         ///     Creation,
26         ///     UpdateOf,
27         ///     UpdateTo,
28         ///     Deletion
29         /// }
30         ///
31         /// private struct Transition
32         /// {
33         ///     public ulong TransactionId;
34         ///     public UniqueTimestamp Timestamp;
35         ///     public TransactionItemType Type;
36         ///     public Link Source;
37         ///     public Link Linker;
38         ///     public Link Target;
39         /// }
40         ///
41         /// Или
42         ///
43         /// public struct TransitionHeader
44         /// {
45         ///     public ulong TransactionIdCombined;
46         ///     public ulong TimestampCombined;
47         ///
48         ///     public ulong TransactionId
49         ///     {
50         ///         get
51         ///         {
52         ///             return (ulong) mask & TransactionIdCombined;
53         ///         }
54         ///     }
55         ///
56         ///     public UniqueTimestamp Timestamp
57         ///     {
58         ///         get
59         ///         {
60         ///             return (UniqueTimestamp)mask & TransactionIdCombined;
61         ///         }
62         ///     }
63         ///
64         ///     public TransactionItemType Type
65         ///     {
66         ///         get
67         ///         {
68             /// // Использовать по одному биту из TransactionId и Timestamp,

```

```

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

```

```

139  ///      3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
140  ↪ операции (трансформации)
141  ///      будут записаны в лог.
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,
189      ↪ Transaction transaction)
190      {
191          layer._currentTransactionId = layer._lastCommittedTransactionId + 1;
192          layer._currentTransactionTransitions = transaction._transitions;
193          layer._currentTransaction = transaction;
194      }
195
196      public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
197      {
198          if (transaction.IsReverted)
199          {
200              throw new InvalidOperationException("Transation is reverted.");
201          }
202          if (transaction.IsCommitted)
203          {
204              throw new InvalidOperationException("Transation is committed.");
205          }
206      }
207
208      protected override void Dispose(bool manual, bool wasDisposed)
209      {
210          if (!wasDisposed && _layer != null && !_layer.IsDisposed)
211          {
212              if (!IsCommitted && !IsReverted)
213              {
214                  Revert();
215              }
216              _layer.ResetCurrentTransation();
217          }
218      }

```

```

216     }
217 }
218
219 public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
220
221 private readonly string _logAddress;
222 private readonly FileStream _log;
223 private readonly Queue<Transition> _transitions;
224 private readonly UniqueTimestampFactory _uniqueTimestampFactory;
225 private Task _transitionsPusher;
226 private Transition _lastCommittedTransition;
227 private ulong _currentTransactionId;
228 private Queue<Transition> _currentTransactionTransitions;
229 private Transaction _currentTransaction;
230 private ulong _lastCommittedTransactionId;
231
232 public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
233     : base(links)
234 {
235     if (string.IsNullOrEmpty(logAddress))
236     {
237         throw new ArgumentNullException(nameof(logAddress));
238     }
239     // В первой строке файла хранится последняя закоммиченную транзакцию.
240     // При запуске это используется для проверки удачного закрытия файла лога.
241     // In the first line of the file the last committed transaction is stored.
242     // On startup, this is used to check that the log file is successfully closed.
243     var lastCommittedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
244     var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
245     if (!lastCommittedTransition.Equals(lastWrittenTransition))
246     {
247         Dispose();
248         throw new NotSupportedException("Database is damaged, autorecovery is not
249             ↳ supported yet.");
250     }
251     if (lastCommittedTransition.Equals(default(Transition)))
252     {
253         FileHelpers.WriteFirst(logAddress, lastCommittedTransition);
254     }
255     _lastCommittedTransition = lastCommittedTransition;
256     // TODO: Think about a better way to calculate or store this value
257     var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
258     _lastCommittedTransactionId = allTransitions.Max(x => x.TransactionId);
259     _uniqueTimestampFactory = new UniqueTimestampFactory();
260     _logAddress = logAddress;
261     _log = FileHelpers.Append(logAddress);
262     _transitions = new Queue<Transition>();
263     _transitionsPusher = new Task(TransitionsPusher);
264     _transitionsPusher.Start();
265 }
266
267 public IList<ulong> GetLinkValue(ulong link) => Links.GetLink(link);
268
269 public override ulong Create()
270 {
271     var createdLinkIndex = Links.Create();
272     var createdLink = new UInt64Link(Links.GetLink(createdLinkIndex));
273     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
274         ↳ default, createdLink));
275     return createdLinkIndex;
276 }
277
278 public override ulong Update(IList<ulong> parts)
279 {
280     var linkIndex = parts[Constants.IndexPart];
281     var beforeLink = new UInt64Link(Links.GetLink(linkIndex));
282     linkIndex = Links.Update(parts);
283     var afterLink = new UInt64Link(Links.GetLink(linkIndex));
284     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
285         ↳ beforeLink, afterLink));
286     return linkIndex;
287 }
288
289 public override void Delete(ulong link)
290 {
291     var deletedLink = new UInt64Link(Links.GetLink(link));
292     Links.Delete(link);
293     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
294         ↳ deletedLink, default));
295 }

```

```

291     }
292
293     [MethodImpl(MethodImplOptions.AggressiveInlining)]
294     private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
        ↳ _transitions;
295
296     private void CommitTransition(Transition transition)
297     {
298         if (_currentTransaction != null)
299         {
300             Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
301         }
302         var transitions = GetCurrentTransitions();
303         transitions.Enqueue(transition);
304     }
305
306     private void RevertTransition(Transition transition)
307     {
308         if (transition.After.IsNull()) // Revert Deletion with Creation
309         {
310             Links.Create();
311         }
312         else if (transition.Before.IsNull()) // Revert Creation with Deletion
313         {
314             Links.Delete(transition.After.Index);
315         }
316         else // Revert Update
317         {
318             Links.Update(new[] { transition.After.Index, transition.Before.Source,
        ↳ transition.Before.Target });
319         }
320     }
321
322     private void ResetCurrentTransation()
323     {
324         _currentTransactionId = 0;
325         _currentTransactionTransitions = null;
326         _currentTransaction = null;
327     }
328
329     private void PushTransitions()
330     {
331         if (_log == null || _transitions == null)
332         {
333             return;
334         }
335         for (var i = 0; i < _transitions.Count; i++)
336         {
337             var transition = _transitions.Dequeue();
338
339             _log.Write(transition);
340             _lastCommittedTransition = transition;
341         }
342     }
343
344     private void TransitionsPusher()
345     {
346         while (!IsDisposed && _transitionsPusher != null)
347         {
348             Thread.Sleep(DefaultPushDelay);
349             PushTransitions();
350         }
351     }
352
353     public Transaction BeginTransaction() => new Transaction(this);
354
355     private void DisposeTransitions()
356     {
357         try
358         {
359             var pusher = _transitionsPusher;
360             if (pusher != null)
361             {
362                 _transitionsPusher = null;
363                 pusher.Wait();
364             }
365             if (_transitions != null)
366             {
367                 PushTransitions();

```

```

368     }
369     _log.DisposeIfPossible();
370     FileHelpers.WriteFirst(_logAddress, _lastCommittedTransition);
371 }
372 catch
373 {
374 }
375 }
376
377 #region DisposalBase
378
379 protected override void Dispose(bool manual, bool wasDisposed)
380 {
381     if (!wasDisposed)
382     {
383         DisposeTransitions();
384     }
385     base.Dispose(manual, wasDisposed);
386 }
387
388 #endregion
389 }
390 }

```

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

```

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

```

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

```

1  using Platform.Data.Doublets.Sequences.Indexes;
2  using Platform.Interfaces;
3  using System.Collections.Generic;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Unicode
8  {
9      public class StringToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
10          ⇨ IConverter<string, TLink>
11     {
12         private readonly IConverter<char, TLink> _charToUnicodeSymbolConverter;
13         private readonly ISequenceIndex<TLink> _index;
14         private readonly IConverter<IList<TLink>, TLink> _listToSequenceLinkConverter;
15         private readonly TLink _unicodeSequenceMarker;
16
17         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
18             ⇨ charToUnicodeSymbolConverter, ISequenceIndex<TLink> index, IConverter<IList<TLink>,
19             ⇨ TLink> listToSequenceLinkConverter, TLink unicodeSequenceMarker) : base(links)
20         {
21             _charToUnicodeSymbolConverter = charToUnicodeSymbolConverter;
22             _index = index;
23             _listToSequenceLinkConverter = listToSequenceLinkConverter;
24             _unicodeSequenceMarker = unicodeSequenceMarker;
25         }
26     }
27 }

```

```

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

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Globalization;
4  using System.Runtime.CompilerServices;
5  using System.Text;
6  using Platform.Data.Sequences;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Unicode
11 {
12     public class UnicodeMap
13     {
14         public static readonly ulong FirstCharLink = 1;
15         public static readonly ulong LastCharLink = FirstCharLink + char.MaxValue;
16         public static readonly ulong MapSize = 1 + char.MaxValue;
17
18         private readonly ILinks<ulong> _links;
19         private bool _initialized;
20
21         public UnicodeMap(ILinks<ulong> links) => _links = links;
22
23         public static UnicodeMap InitNew(ILinks<ulong> links)
24         {
25             var map = new UnicodeMap(links);
26             map.Init();
27             return map;
28         }
29
30         public void Init()
31         {
32             if (_initialized)
33             {
34                 return;
35             }
36             _initialized = true;
37             var firstLink = _links.CreatePoint();
38             if (firstLink != FirstCharLink)
39             {
40                 _links.Delete(firstLink);
41             }
42             else
43             {
44                 for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
45                 {
46                     // From NIL to It (NIL -> Character) transformation meaning, (or infinite
47                     ↪ amount of NIL characters before actual Character)
48                     var createdLink = _links.CreatePoint();
49                     _links.Update(createdLink, firstLink, createdLink);
50                     if (createdLink != i)
51                     {
52                         throw new InvalidOperationException("Unable to initialize UTF 16
53                         ↪ table.");
54                     }
55                 }
56             }
57         }
58
59         // 0 - null link
60         // 1 - nil character (0 character)
61         // ...
62         // 65536 (0(1) + 65535 = 65536 possible values)
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         public static ulong FromCharToLink(char character) => (ulong)character + 1;

```



```

64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 public static char FromLinkToChar(ulong link) => (char)(link - 1);
66
67 [MethodImpl(MethodImplOptions.AggressiveInlining)]
68 public static bool IsCharLink(ulong link) => link <= MapSize;
69
70 public static string FromLinksToString(IList<ulong> linksList)
71 {
72     var sb = new StringBuilder();
73     for (int i = 0; i < linksList.Count; i++)
74     {
75         sb.Append(FromLinkToChar(linksList[i]));
76     }
77     return sb.ToString();
78 }
79
80 public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
81 {
82     var sb = new StringBuilder();
83     if (links.Exists(link))
84     {
85         StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
86             x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
87             ↪ element =>
88             {
89                 sb.Append(FromLinkToChar(element));
90                 return true;
91             });
92     }
93     return sb.ToString();
94 }
95
96 public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
97     ↪ chars.Length);
98
99 public static ulong[] FromCharsToLinkArray(char[] chars, int count)
100 {
101     // char array to ulong array
102     var linksSequence = new ulong[count];
103     for (var i = 0; i < count; i++)
104     {
105         linksSequence[i] = FromCharToLink(chars[i]);
106     }
107     return linksSequence;
108 }
109
110 public static ulong[] FromStringToLinkArray(string sequence)
111 {
112     // char array to ulong array
113     var linksSequence = new ulong[sequence.Length];
114     for (var i = 0; i < sequence.Length; i++)
115     {
116         linksSequence[i] = FromCharToLink(sequence[i]);
117     }
118     return linksSequence;
119 }
120
121 public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
122 {
123     var result = new List<ulong[]>();
124     var offset = 0;
125     while (offset < sequence.Length)
126     {
127         var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
128         var relativeLength = 1;
129         var absoluteLength = offset + relativeLength;
130         while (absoluteLength < sequence.Length &&
131             currentCategory ==
132             ↪ CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
133         {
134             relativeLength++;
135             absoluteLength++;
136         }
137         // char array to ulong array
138         var innerSequence = new ulong[relativeLength];
139         var maxLength = offset + relativeLength;
140         for (var i = offset; i < maxLength; i++)
141         {

```

```

140         innerSequence[i - offset] = FromCharToLink(sequence[i]);
141     }
142     result.Add(innerSequence);
143     offset += relativeLength;
144 }
145 return result;
146 }
147
148 public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
149 {
150     var result = new List<ulong[]>();
151     var offset = 0;
152     while (offset < array.Length)
153     {
154         var relativeLength = 1;
155         if (array[offset] <= LastCharLink)
156         {
157             var currentCategory =
158                 ↪ CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
159             var absoluteLength = offset + relativeLength;
160             while (absoluteLength < array.Length &&
161                 array[absoluteLength] <= LastCharLink &&
162                 currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(
163                     ↪ array[absoluteLength])))
164             {
165                 relativeLength++;
166                 absoluteLength++;
167             }
168         }
169         else
170         {
171             var absoluteLength = offset + relativeLength;
172             while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
173             {
174                 relativeLength++;
175                 absoluteLength++;
176             }
177             // copy array
178             var innerSequence = new ulong[relativeLength];
179             var maxLength = offset + relativeLength;
180             for (var i = offset; i < maxLength; i++)
181             {
182                 innerSequence[i - offset] = array[i];
183             }
184             result.Add(innerSequence);
185             offset += relativeLength;
186         }
187     }
188     return result;
189 }

```

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

```

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

```

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

```

1 using System;
2 using System.Linq;
3 using Platform.Data.Doublets.Sequences.Walkers;
4 using Platform.Interfaces;

```

```

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

```

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

```

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

```

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

```

1 using System;
2 using Platform.Interfaces;
3 using Platform.Numbers;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Unicode
8 {
9     public class UnicodeSymbolToCharConverter<TLink> : LinksOperatorBase<TLink>,
10         ↳ IConverter<TLink, char>
11     {
12         private readonly IConverter<TLink> _numberToAddressConverter;
13         private readonly ICriterionMatcher<TLink> _unicodeSymbolCriterionMatcher;
14
15         public UnicodeSymbolToCharConverter(ILinks<TLink> links, IConverter<TLink>
16             ↳ numberToAddressConverter, ICriterionMatcher<TLink> unicodeSymbolCriterionMatcher) :
17             ↳ base(links)
18         {
19             _numberToAddressConverter = numberToAddressConverter;
20             _unicodeSymbolCriterionMatcher = unicodeSymbolCriterionMatcher;
21         }
22     }
23 }

```

```

19
20 public char Convert(TLink source)
21 {
22     if (!_unicodeSymbolCriterionMatcher.IsMatched(source))
23     {
24         throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
        ↳ not a unicode symbol.");
25     }
26     return (char)(ushort)(Integer<TLink>)_numberToAddressConverter.Convert(Links.GetSource
        ↳ ce(source));
27 }
28 }
29 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using Xunit;
4 using Platform.Diagnostics;
5
6 namespace Platform.Data.Doublets.Tests
7 {
8     public static class ComparisonTests
9     {
10         protected class UInt64Comparer : IComparer<ulong>
11         {
12             public int Compare(ulong x, ulong y) => x.CompareTo(y);
13         }
14
15         private static int Compare(ulong x, ulong y) => x.CompareTo(y);
16
17         [Fact]
18         public static void GreaterOrEqualPerformanceTest()
19         {
20             const int N = 1000000;
21
22             ulong x = 10;
23             ulong y = 500;
24
25             bool result = false;
26
27             var ts1 = Performance.Measure(() =>
28             {
29                 for (int i = 0; i < N; i++)
30                 {
31                     result = Compare(x, y) >= 0;
32                 }
33             });
34
35             var comparer1 = Comparer<ulong>.Default;
36
37             var ts2 = Performance.Measure(() =>
38             {
39                 for (int i = 0; i < N; i++)
40                 {
41                     result = comparer1.Compare(x, y) >= 0;
42                 }
43             });
44
45             Func<ulong, ulong, int> compareReference = comparer1.Compare;
46
47             var ts3 = Performance.Measure(() =>
48             {
49                 for (int i = 0; i < N; i++)
50                 {
51                     result = compareReference(x, y) >= 0;
52                 }
53             });
54
55             var comparer2 = new UInt64Comparer();
56
57             var ts4 = Performance.Measure(() =>
58             {
59                 for (int i = 0; i < N; i++)
60                 {
61                     result = comparer2.Compare(x, y) >= 0;
62                 }
63             });
64
65             Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {result}");

```

```

66     }
67 }
68 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using Xunit;
4  using Platform.Diagnostics;
5
6  namespace Platform.Data.Doublets.Tests
7  {
8      public static class EqualityTests
9      {
10         protected class UInt64EqualityComparer : IEqualityComparer<ulong>
11         {
12             public bool Equals(ulong x, ulong y) => x == y;
13
14             public int GetHashCode(ulong obj) => obj.GetHashCode();
15         }
16
17         private static bool Equals1<T>(T x, T y) => Equals(x, y);
18
19         private static bool Equals2<T>(T x, T y) => x.Equals(y);
20
21         private static bool Equals3(ulong x, ulong y) => x == y;
22
23         [Fact]
24         public static void EqualsPerformanceTest()
25         {
26             const int N = 1000000;
27
28             ulong x = 10;
29             ulong y = 500;
30
31             bool result = false;
32
33             var ts1 = Performance.Measure(() =>
34             {
35                 for (int i = 0; i < N; i++)
36                 {
37                     result = Equals1(x, y);
38                 }
39             });
40
41             var ts2 = Performance.Measure(() =>
42             {
43                 for (int i = 0; i < N; i++)
44                 {
45                     result = Equals2(x, y);
46                 }
47             });
48
49             var ts3 = Performance.Measure(() =>
50             {
51                 for (int i = 0; i < N; i++)
52                 {
53                     result = Equals3(x, y);
54                 }
55             });
56
57             var equalityComparer1 = EqualityComparer<ulong>.Default;
58
59             var ts4 = Performance.Measure(() =>
60             {
61                 for (int i = 0; i < N; i++)
62                 {
63                     result = equalityComparer1.Equals(x, y);
64                 }
65             });
66
67             var equalityComparer2 = new UInt64EqualityComparer();
68
69             var ts5 = Performance.Measure(() =>
70             {
71                 for (int i = 0; i < N; i++)
72                 {
73                     result = equalityComparer2.Equals(x, y);
74                 }
75             });

```

```

76
77     Func<ulong, ulong, bool> equalityComparer3 = equalityComparer2.Equals;
78
79     var ts6 = Performance.Measure(() =>
80     {
81         for (int i = 0; i < N; i++)
82         {
83             result = equalityComparer3(x, y);
84         }
85     });
86
87     var comparer = Comparer<ulong>.Default;
88
89     var ts7 = Performance.Measure(() =>
90     {
91         for (int i = 0; i < N; i++)
92         {
93             result = comparer.Compare(x, y) == 0;
94         }
95     });
96
97     Assert.True(ts2 < ts1);
98     Assert.True(ts3 < ts2);
99     Assert.True(ts5 < ts4);
100    Assert.True(ts5 < ts6);
101
102    Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {ts5} {ts6} {ts7} {result}");
103
104 }
105 }

```

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

```

1  using System;
2  using System.Runtime.InteropServices;
3  using Xunit;
4  using Platform.Reflection;
5  using Platform.Memory;
6  using Platform.Scopes;
7  using Platform.Data.Doublets.ResizableDirectMemory;
8
9  namespace Platform.Data.Doublets.Tests
10 {
11     public static class GenericLinksTests
12     {
13         [Fact]
14         public static void CRUDTest()
15         {
16             Using<byte>(links => links.TestCRUDOperations());
17             Using<ushort>(links => links.TestCRUDOperations());
18             Using<uint>(links => links.TestCRUDOperations());
19             Using<ulong>(links => links.TestCRUDOperations());
20         }
21
22         [Fact]
23         public static void RawNumbersCRUDTest()
24         {
25             Using<byte>(links => links.TestRawNumbersCRUDOperations());
26             Using<ushort>(links => links.TestRawNumbersCRUDOperations());
27             Using<uint>(links => links.TestRawNumbersCRUDOperations());
28             Using<ulong>(links => links.TestRawNumbersCRUDOperations());
29         }
30
31         [Fact]
32         public static void MultipleRandomCreationsAndDeletionsTest()
33         {
34             if (!RuntimeInformation.IsOSPlatform(OSPlatform.Linux))
35             {
36                 Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().
37                     ↪ TestMultipleRandomCreationsAndDeletions(16)); // Cannot use more because
38                     ↪ current implementation of tree cuts out 5 bits from the address space.
39                 Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().
40                     ↪ TestMultipleRandomCreationsAndDeletions(100));
41                 Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().
42                     ↪ TestMultipleRandomCreationsAndDeletions(100));
43             }
44             Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
45                 ↪ tMultipleRandomCreationsAndDeletions(100));
46         }
47     }
48 }

```

```

43     private static void Using<TLink>(Action<ILinks<TLink>> action)
44     {
45         using (var scope = new Scope<Types<HeapResizableDirectMemory,
46             ↳ ResizableDirectMemoryLinks<TLink>>>())
47         {
48             action(scope.Use<ILinks<TLink>>());
49         }
50     }
51 }

```

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

```

1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4  using Xunit;
5  using Platform.Data.Doublets.Sequences;
6  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
7  using Platform.Data.Doublets.Sequences.Frequencies.Counters;
8  using Platform.Data.Doublets.Sequences.Converters;
9  using Platform.Data.Doublets.PropertyOperators;
10 using Platform.Data.Doublets.Incrementers;
11 using Platform.Data.Doublets.Sequences.Walkers;
12 using Platform.Data.Doublets.Sequences.Indexes;
13 using Platform.Data.Doublets.Unicode;
14 using Platform.Data.Doublets.Numbers.Unary;
15
16 namespace Platform.Data.Doublets.Tests
17 {
18     public static class OptimalVariantSequenceTests
19     {
20         private const string SequenceExample = "зеленела зелёная зелень";
21
22         [Fact]
23         public static void LinksBasedFrequencyStoredOptimalVariantSequenceTest()
24         {
25             using (var scope = new TempLinksTestScope(useSequences: false))
26             {
27                 var links = scope.Links;
28                 var constants = links.Constants;
29
30                 links.UseUnicode();
31
32                 var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);
33
34                 var meaningRoot = links.CreatePoint();
35                 var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
36                 var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
37                 var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
38                     ↳ constants.Itself);
39
40                 var unaryNumberToAddressConverter = new
41                     ↳ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
42                 var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
43                 var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
44                     ↳ frequencyMarker, unaryOne, unaryNumberIncrementer);
45                 var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
46                     ↳ frequencyPropertyMarker, frequencyMarker);
47                 var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
48                     ↳ frequencyPropertyOperator, frequencyIncrementer);
49                 var linkToItsFrequencyNumberConverter = new
50                     ↳ LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
51                     ↳ unaryNumberToAddressConverter);
52                 var sequenceToItsLocalElementLevelsConverter = new
53                     ↳ SequenceToItsLocalElementLevelsConverter<ulong>(links,
54                     ↳ linkToItsFrequencyNumberConverter);
55                 var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
56                     ↳ sequenceToItsLocalElementLevelsConverter);
57
58                 var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
59                     ↳ Walker = new LeveledSequenceWalker<ulong>(links) });
60
61                 ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
62                     ↳ index, optimalVariantConverter);
63             }
64
65             [Fact]
66             public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()

```

```

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
68                 ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links);
69
70             var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
71                 ↪ totalSequenceSymbolFrequencyCounter);
72
73             var index = new
74                 ↪ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
75             var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache);
76
77             var sequenceToItsLocalElementLevelsConverter = new
78                 ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
79                 ↪ linkToItsFrequencyNumberConverter);
80             var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
81                 ↪ sequenceToItsLocalElementLevelsConverter);
82
83             var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
84                 ↪ Walker = new LeveledSequenceWalker<ulong>(links) });
85
86             ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
87                 ↪ index, optimalVariantConverter);
88         }
89     }
90
91     private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
92         ↪ SequenceToItsLocalElementLevelsConverter<ulong>
93         ↪ sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
94         ↪ OptimalVariantConverter<ulong> optimalVariantConverter)
95     {
96         index.Add(sequence);
97
98         var optimalVariant = optimalVariantConverter.Convert(sequence);
99
100        var readSequence1 = sequences.ToList(optimalVariant);
101
102        Assert.True(sequence.SequenceEqual(readSequence1));
103    }
104 }

```

./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<ulong>() {
24                     ↪ Walker = new LeveledSequenceWalker<ulong>(links) });
25
26                 var sequence = new ulong[sequenceLength];
27                 for (var i = 0; i < sequenceLength; i++)

```



```

27     {
28         sequence[i] = links.Create();
29     }
30
31     var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
32
33     var sw1 = Stopwatch.StartNew();
34     var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
35
36     var sw2 = Stopwatch.StartNew();
37     var readSequence1 = sequences.ToList(balancedVariant); sw2.Stop();
38
39     var sw3 = Stopwatch.StartNew();
40     var readSequence2 = new List<ulong>();
41     SequenceWalker.WalkRight(balancedVariant,
42                             links.GetSource,
43                             links.GetTarget,
44                             links.IsPartialPoint,
45                             readSequence2.Add);
46
47     sw3.Stop();
48
49     Assert.True(sequence.SequenceEqual(readSequence1));
50
51     Assert.True(sequence.SequenceEqual(readSequence2));
52
53     // Assert.True(sw2.Elapsed < sw3.Elapsed);
54
55     Console.WriteLine($"Stack-based walker: {sw3.Elapsed}, Level-based reader:
56     ↪ {sw2.Elapsed}");
57
58     for (var i = 0; i < sequenceLength; i++)
59     {
60         links.Delete(sequence[i]);
61     }
62 }
63 }

```

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

```

1  using System.IO;
2  using Xunit;
3  using Platform.Singletons;
4  using Platform.Memory;
5  using Platform.Data.Constants;
6  using Platform.Data.Doublets.ResizableDirectMemory;
7
8  namespace Platform.Data.Doublets.Tests
9  {
10     public static class ResizableDirectMemoryLinksTests
11     {
12         private static readonly LinksCombinedConstants<ulong, ulong, int> _constants =
13             ↪ Default<LinksCombinedConstants<ulong, ulong, int>>.Instance;
14
15         [Fact]
16         public static void BasicFileMappedMemoryTest()
17         {
18             var tempFilename = Path.GetTempFileName();
19             using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(tempFilename))
20             {
21                 memoryAdapter.TestBasicMemoryOperations();
22             }
23             File.Delete(tempFilename);
24
25             [Fact]
26             public static void BasicHeapMemoryTest()
27             {
28                 using (var memory = new
29                     ↪ HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
30                 using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
31                     ↪ UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
32                 {
33                     memoryAdapter.TestBasicMemoryOperations();
34                 }
35
36                 private static void TestBasicMemoryOperations(this ILinks<ulong> memoryAdapter)
37                 {

```

```

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     }

```

./Platform.Data.Doublets.Tests/ScopeTests.cs

```

1  using Xunit;
2  using Platform.Scopes;
3  using Platform.Memory;
4  using Platform.Data.Doublets.ResizableDirectMemory;
5  using Platform.Data.Doublets.Decorators;
6
7  namespace Platform.Data.Doublets.Tests
8  {
9      public static class ScopeTests
10     {
11         [Fact]
12         public static void SingleDependencyTest()
13         {
14             using (var scope = new Scope())
15             {
16                 scope.IncludeAssemblyOf<IMemory>();
17                 var instance = scope.Use<IDirectMemory>();
18                 Assert.IsType<HeapResizableDirectMemory>(instance);
19             }
20         }
21
22         [Fact]
23         public static void CascadeDependencyTest()
24         {
25             using (var scope = new Scope())
26             {
27                 scope.Include<TemporaryFileMappedResizableDirectMemory>();
28                 scope.Include<UInt64ResizableDirectMemoryLinks>();
29                 var instance = scope.Use<ILinks<ulong>>();
30                 Assert.IsType<UInt64ResizableDirectMemoryLinks>(instance);
31             }
32         }
33
34         [Fact]
35         public static void FullAutoResolutionTest()
36         {
37             using (var scope = new Scope(autoInclude: true, autoExplore: true))
38             {
39                 var instance = scope.Use<UInt64Links>();
40                 Assert.IsType<UInt64Links>(instance);
41             }
42         }
43     }
44 }

```

./Platform.Data.Doublets.Tests/SequencesTests.cs

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

```

79 //         SequencesOptions o = new SequencesOptions();
80
81 // TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
82 //         o.
83
84 //         var sequences = new Sequences(links);
85
86 //         var sw1 = Stopwatch.StartNew();
87 //         var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
88
89 //         var sw2 = Stopwatch.StartNew();
90 //         var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
91
92 //         Assert.True(results1.Count > results2.Length);
93 //         Assert.True(sw1.Elapsed > sw2.Elapsed);
94
95 //         for (var i = 0; i < sequenceLength; i++)
96 //             links.Delete(sequence[i]);
97 //     }
98
99 //     File.Delete(tempFilename);
100 // }
101
102 [Fact]
103 public static void AllVariantsSearchTest()
104 {
105     const long sequenceLength = 8;
106
107     using (var scope = new TempLinksTestScope(useSequences: true))
108     {
109         var links = scope.Links;
110         var sequences = scope.Sequences;
111
112         var sequence = new ulong[sequenceLength];
113         for (var i = 0; i < sequenceLength; i++)
114         {
115             sequence[i] = links.Create();
116         }
117
118         var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();
119
120         //for (int i = 0; i < createResults.Length; i++)
121         //    sequences.Create(createResults[i]);
122
123         var sw0 = Stopwatch.StartNew();
124         var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
125
126         var sw1 = Stopwatch.StartNew();
127         var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
128
129         var sw2 = Stopwatch.StartNew();
130         var searchResults2 = sequences.Each1(sequence); sw2.Stop();
131
132         var sw3 = Stopwatch.StartNew();
133         var searchResults3 = sequences.Each(sequence); 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
158 [Fact]

```

```

159 public static void BalancedVariantSearchTest()
160 {
161     const long sequenceLength = 200;
162
163     using (var scope = new TempLinksTestScope(useSequences: true))
164     {
165         var links = scope.Links;
166         var sequences = scope.Sequences;
167
168         var sequence = new ulong[sequenceLength];
169         for (var i = 0; i < sequenceLength; i++)
170         {
171             sequence[i] = links.Create();
172         }
173
174         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
175
176         var sw1 = Stopwatch.StartNew();
177         var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
178
179         var sw2 = Stopwatch.StartNew();
180         var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();
181
182         var sw3 = Stopwatch.StartNew();
183         var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();
184
185         // На количестве в 200 элементов это будет занимать вечность
186         //var sw4 = Stopwatch.StartNew();
187         //var searchResults4 = sequences.Each(sequence); sw4.Stop();
188
189         Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);
190
191         Assert.True(searchResults3.Count == 1 && balancedVariant ==
192             ↪ searchResults3.First());
193
194         //Assert.True(sw1.Elapsed < sw2.Elapsed);
195
196         for (var i = 0; i < sequenceLength; i++)
197         {
198             links.Delete(sequence[i]);
199         }
200     }
201
202     [Fact]
203     public static void AllPartialVariantsSearchTest()
204     {
205         const long sequenceLength = 8;
206
207         using (var scope = new TempLinksTestScope(useSequences: true))
208         {
209             var links = scope.Links;
210             var sequences = scope.Sequences;
211
212             var sequence = new ulong[sequenceLength];
213             for (var i = 0; i < sequenceLength; i++)
214             {
215                 sequence[i] = links.Create();
216             }
217
218             var createResults = sequences.CreateAllVariants2(sequence);
219
220             //var createResultsStrings = createResults.Select(x => x + ": " +
221             ↪ sequences.FormatSequence(x)).ToList();
222             //Global.Trash = createResultsStrings;
223
224             var partialSequence = new ulong[sequenceLength - 2];
225
226             Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
227
228             var sw1 = Stopwatch.StartNew();
229             var searchResults1 =
230             ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
231
232             var sw2 = Stopwatch.StartNew();
233             var searchResults2 =
234             ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
235
236             //var sw3 = Stopwatch.StartNew();

```

```

234         //var searchResults3 =
235         ↪ sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
236
237     var sw4 = Stopwatch.StartNew();
238     var searchResults4 =
239     ↪ sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();
240
241     //Global.Trash = searchResults3;
242
243     //var searchResults1Strings = searchResults1.Select(x => x + ": " +
244     ↪ sequences.FormatSequence(x)).ToList();
245     //Global.Trash = searchResults1Strings;
246
247     var intersection1 = createResults.Intersect(searchResults1).ToList();
248     Assert.True(intersection1.Count == createResults.Length);
249
250     var intersection2 = createResults.Intersect(searchResults2).ToList();
251     Assert.True(intersection2.Count == createResults.Length);
252
253     var intersection4 = createResults.Intersect(searchResults4).ToList();
254     Assert.True(intersection4.Count == createResults.Length);
255
256     for (var i = 0; i < sequenceLength; i++)
257     {
258         links.Delete(sequence[i]);
259     }
260 }
261
262 [Fact]
263 public static void BalancedPartialVariantsSearchTest()
264 {
265     const long sequenceLength = 200;
266
267     using (var scope = new TempLinksTestScope(useSequences: true))
268     {
269         var links = scope.Links;
270         var sequences = scope.Sequences;
271
272         var sequence = new ulong[sequenceLength];
273         for (var i = 0; i < sequenceLength; i++)
274         {
275             sequence[i] = links.Create();
276         }
277
278         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
279         var balancedVariant = balancedVariantConverter.Convert(sequence);
280
281         var partialSequence = new ulong[sequenceLength - 2];
282         Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
283
284         var sw1 = Stopwatch.StartNew();
285         var searchResults1 =
286         ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
287
288         var sw2 = Stopwatch.StartNew();
289         var searchResults2 =
290         ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
291
292         Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
293         Assert.True(searchResults2.Count == 1 && balancedVariant ==
294         ↪ searchResults2.First());
295
296         for (var i = 0; i < sequenceLength; i++)
297         {
298             links.Delete(sequence[i]);
299         }
300     }
301 }
302
303 [Fact(Skip = "Correct implementation is pending")]
304 public static void PatternMatchTest()
305 {
306     var zeroOrMany = Sequences.Sequences.ZeroOrMany;
307
308     using (var scope = new TempLinksTestScope(useSequences: true))

```

```

307     {
308         var links = scope.Links;
309         var sequences = scope.Sequences;
310
311         var e1 = links.Create();
312         var e2 = links.Create();
313
314         var sequence = new[]
315         {
316             e1, e2, e1, e2 // mama / papa
317         };
318
319         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
320
321         var balancedVariant = balancedVariantConverter.Convert(sequence);
322
323         // 1: [1]
324         // 2: [2]
325         // 3: [1,2]
326         // 4: [1,2,1,2]
327
328         var doublet = links.GetSource(balancedVariant);
329
330         var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);
331
332         Assert.True(matchedSequences1.Count == 0);
333
334         var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);
335
336         Assert.True(matchedSequences2.Count == 0);
337
338         var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);
339
340         Assert.True(matchedSequences3.Count == 0);
341
342         var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
343
344         Assert.Contains(doublet, matchedSequences4);
345         Assert.Contains(balancedVariant, matchedSequences4);
346
347         for (var i = 0; i < sequence.Length; i++)
348         {
349             links.Delete(sequence[i]);
350         }
351     }
352 }
353
354 [Fact]
355 public static void IndexTest()
356 {
357     using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
358         ↳ true }, useSequences: true))
359     {
360         var links = scope.Links;
361         var sequences = scope.Sequences;
362         var index = sequences.Options.Index;
363
364         var e1 = links.Create();
365         var e2 = links.Create();
366
367         var sequence = new[]
368         {
369             e1, e2, e1, e2 // mama / papa
370         };
371
372         Assert.False(index.MightContain(sequence));
373
374         index.Add(sequence);
375
376         Assert.True(index.MightContain(sequence));
377     }
378 }
379
380 /// <summary>Imported from https://raw.githubusercontent.com/wiki/Konard/LinksPlatform/%
381 ↳ 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
382 ↳ %D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>
383 private static readonly string _exampleText =
384     @"([english
385     ↳ version] (https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))

```

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

384

385 [![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)

386

387 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?

388

389 [![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)

390

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

392

393 [![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)

394

395 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?

396

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

398

399 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?

400

401 [![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

403 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?

404

405 [![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)

406

407 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?

408

409 [![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)

410

411 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?

412

413 [![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)

414


```

415 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?
416
417 [![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)
418
419 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?
420
421 [![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)
422
423 На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом
    ↳ рекурсии или фрактала?
424
425 [![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)
426
427 Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы? Буквы?
    ↳ Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?
428
429 [![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)
430
431 ...
432
433 [![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-anim
    ↳ ation-500.gif
    ↳ ""анимация"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro
    ↳ -animation-500.gif)";
434
435     private static readonly string _exampleLoremIpsumText =
436         @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
            ↳ incididunt ut labore et dolore magna aliqua.
437 Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
    ↳ consequat.";
438
439     [Fact]
440     public static void CompressionTest()
441     {
442         using (var scope = new TempLinksTestScope(useSequences: true))
443         {
444             var links = scope.Links;
445             var sequences = scope.Sequences;
446
447             var e1 = links.Create();
448             var e2 = links.Create();
449
450             var sequence = new[]
451             {
452                 e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
453             };
454
455             var balancedVariantConverter = new BalancedVariantConverter<ulong>(links.Unsync);
456             var totalSequenceSymbolFrequencyCounter = new
                ↳ TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
457             var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync,
                ↳ totalSequenceSymbolFrequencyCounter);
458             var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
                ↳ balancedVariantConverter, doubletFrequenciesCache);
459

```

```

460     var compressedVariant = compressingConverter.Convert(sequence);
461
462     // 1: [1]          (1->1) point
463     // 2: [2]          (2->2) point
464     // 3: [1,2]        (1->2) doublet
465     // 4: [1,2,1,2]    (3->3) doublet
466
467     Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
468     Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
469     Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
470     Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);
471
472     var source = _constants.SourcePart;
473     var target = _constants.TargetPart;
474
475     Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
476     Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
477     Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
478     Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);
479
480     // 4 - length of sequence
481     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0)
482         ↪ == sequence[0]);
483     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
484         ↪ == sequence[1]);
485     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
486         ↪ == sequence[2]);
487     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
488         ↪ == sequence[3]);
489 }
490
491 [Fact]
492 public static void CompressionEfficiencyTest()
493 {
494     var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
495         ↪ StringSplitOptions.RemoveEmptyEntries);
496     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
497     var totalCharacters = arrays.Select(x => x.Length).Sum();
498
499     using (var scope1 = new TempLinksTestScope(useSequences: true))
500     using (var scope2 = new TempLinksTestScope(useSequences: true))
501     using (var scope3 = new TempLinksTestScope(useSequences: true))
502     {
503         scope1.Links.Unsync.UseUnicode();
504         scope2.Links.Unsync.UseUnicode();
505         scope3.Links.Unsync.UseUnicode();
506
507         var balancedVariantConverter1 = new
508             ↪ BalancedVariantConverter<ulong>(scope1.Links.Unsync);
509         var totalSequenceSymbolFrequencyCounter = new
510             ↪ TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
511         var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
512             ↪ totalSequenceSymbolFrequencyCounter);
513         var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
514             ↪ balancedVariantConverter1, linkFrequenciesCache1,
515             ↪ doInitialFrequenciesIncrement: false);
516
517         var compressor2 = scope2.Sequences;
518         var compressor3 = scope3.Sequences;
519
520         var constants = Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
521
522         var sequences = compressor3;
523         //var meaningRoot = links.CreatePoint();
524         //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
525         //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
526         //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
527             ↪ constants.Itself);
528
529         //var unaryNumberToAddressConverter = new
530             ↪ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
531         //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
532             ↪ unaryOne);
533         //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
534             ↪ frequencyMarker, unaryOne, unaryNumberIncrementer);
535         //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
536             ↪ frequencyPropertyMarker, frequencyMarker);

```

```

523 //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
    ↳ frequencyPropertyOperator, frequencyIncrementer);
524 //var linkToItsFrequencyNumberConverter = new
    ↳ LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
    ↳ unaryNumberToAddressConverter);
525
526 var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
    ↳ totalSequenceSymbolFrequencyCounter);
527
528 var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
    ↳ ncyNumberConverter<ulong>(linkFrequenciesCache3);
529
530 var sequenceToItsLocalElementLevelsConverter = new
    ↳ SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
    ↳ linkToItsFrequencyNumberConverter);
531 var optimalVariantConverter = new
    ↳ OptimalVariantConverter<ulong>(scope3.Links.Unsync,
    ↳ sequenceToItsLocalElementLevelsConverter);
532
533 var compressed1 = new ulong[arrays.Length];
534 var compressed2 = new ulong[arrays.Length];
535 var compressed3 = new ulong[arrays.Length];
536
537 var START = 0;
538 var END = arrays.Length;
539
540 //for (int i = START; i < END; i++)
541 //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
542
543 var initialCount1 = scope2.Links.Unsync.Count();
544
545 var sw1 = Stopwatch.StartNew();
546
547 for (int i = START; i < END; i++)
548 {
549     linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
550     compressed1[i] = compressor1.Convert(arrays[i]);
551 }
552
553 var elapsed1 = sw1.Elapsed;
554
555 var balancedVariantConverter2 = new
    ↳ BalancedVariantConverter<ulong>(scope2.Links.Unsync);
556
557 var initialCount2 = scope2.Links.Unsync.Count();
558
559 var sw2 = Stopwatch.StartNew();
560
561 for (int i = START; i < END; i++)
562 {
563     compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
564 }
565
566 var elapsed2 = sw2.Elapsed;
567
568 for (int i = START; i < END; i++)
569 {
570     linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
571 }
572
573 var initialCount3 = scope3.Links.Unsync.Count();
574
575 var sw3 = Stopwatch.StartNew();
576
577 for (int i = START; i < END; i++)
578 {
579     //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
580     compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
581 }
582
583 var elapsed3 = sw3.Elapsed;
584
585 Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
    ↳ 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($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
631     ↪ totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) /
632     ↪ totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) /
633     ↪ totalCharacters}");
634
635 Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
636     ↪ scope2.Links.Unsync.Count() - initialCount2);
637 Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
638     ↪ scope2.Links.Unsync.Count() - initialCount2);
639
640 var duplicateProvider1 = new
641     ↪ DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
642 var duplicateProvider2 = new
643     ↪ DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
644 var duplicateProvider3 = new
645     ↪ DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
646
647 var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
648 var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
649 var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
650
651 var duplicates1 = duplicateCounter1.Count();
652
653 ConsoleHelpers.Debug("-----");
654
655 var duplicates2 = duplicateCounter2.Count();
656
657 ConsoleHelpers.Debug("-----");
658
659 var duplicates3 = duplicateCounter3.Count();
660
661 Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
662
663 linkFrequenciesCache1.ValidateFrequencies();
664 linkFrequenciesCache3.ValidateFrequencies();
665
666 }
667
668 }
669

```

```

650 public static void CompressionStabilityTest()
651 {
652     // TODO: Fix bug (do a separate test)
653     //const ulong minNumbers = 0;
654     //const ulong maxNumbers = 1000;
655
656     const ulong minNumbers = 10000;
657     const ulong maxNumbers = 12500;
658
659     var strings = new List<string>();
660
661     for (ulong i = minNumbers; i < maxNumbers; i++)
662     {
663         strings.Add(i.ToString());
664     }
665
666     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
667     var totalCharacters = arrays.Select(x => x.Length).Sum();
668
669     using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
        ↳ SequencesOptions<ulong> { UseCompression = true,
        ↳ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
        using (var scope2 = new TempLinksTestScope(useSequences: true))
        {
670         scope1.Links.UseUnicode();
671         scope2.Links.UseUnicode();
672
673         //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
674         var compressor1 = scope1.Sequences;
675         var compressor2 = scope2.Sequences;
676
677         var compressed1 = new ulong[arrays.Length];
678         var compressed2 = new ulong[arrays.Length];
679
680         var sw1 = Stopwatch.StartNew();
681
682         var START = 0;
683         var END = arrays.Length;
684
685         // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
686         // Stability issue starts at 10001 or 11000
687         //for (int i = START; i < END; i++)
688         //{
689             // var first = compressor1.Compress(arrays[i]);
690             // var second = compressor1.Compress(arrays[i]);
691
692             // if (first == second)
693             //     compressed1[i] = first;
694             // else
695             // {
696                 // TODO: Find a solution for this case
697             // }
698         //}
699
700         for (int i = START; i < END; i++)
701         {
702             var first = compressor1.Create(arrays[i]);
703             var second = compressor1.Create(arrays[i]);
704
705             if (first == second)
706             {
707                 compressed1[i] = first;
708             }
709             else
710             {
711                 // TODO: Find a solution for this case
712             }
713         }
714
715         var elapsed1 = sw1.Elapsed;
716
717         var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
718
719         var sw2 = Stopwatch.StartNew();
720
721         for (int i = START; i < END; i++)
722         {
723             var first = balancedVariantConverter.Convert(arrays[i]);
724             var second = balancedVariantConverter.Convert(arrays[i]);
725
726
727

```

```

728         if (first == second)
729         {
730             compressed2[i] = first;
731         }
732     }
733
734     var elapsed2 = sw2.Elapsed;
735
736     Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
737     ↪ {elapsed2}");
738
739     Assert.True(elapsed1 > elapsed2);
740
741     // Checks
742     for (int i = START; i < END; i++)
743     {
744         var sequence1 = compressed1[i];
745         var sequence2 = compressed2[i];
746
747         if (sequence1 != _constants.Null && sequence2 != _constants.Null)
748         {
749             var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
750             ↪ scope1.Links);
751
752             var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
753             ↪ scope2.Links);
754
755             //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
756             ↪ link.IsPartialPoint());
757             //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
758             ↪ link.IsPartialPoint());
759
760             //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
761             ↪ arrays[i].Length > 3)
762             //    Assert.False(structure1 == structure2);
763
764             Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
765         }
766     }
767
768     Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
769     Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
770
771     Debug.WriteLine($"{{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
772     ↪ totalCharacters}} | {{(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
773     ↪ totalCharacters}}");
774
775     Assert.True(scope1.Links.Count() <= scope2.Links.Count());
776
777     //compressor1.ValidateFrequencies();
778 }
779
780 [Fact]
781 public static void RandomNumbersCompressionQualityTest()
782 {
783     const ulong N = 500;
784
785     //const ulong minNumbers = 10000;
786     //const ulong maxNumbers = 20000;
787
788     //var strings = new List<string>();
789
790     //for (ulong i = 0; i < N; i++)
791     //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
792     ↪ maxNumbers).ToString());
793
794     var strings = new List<string>();
795
796     for (ulong i = 0; i < N; i++)
797     {
798         strings.Add(RandomHelpers.Default.NextUInt64().ToString());
799     }
800
801     strings = strings.Distinct().ToList();
802
803     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
804     var totalCharacters = arrays.Select(x => x.Length).Sum();

```

```

798 using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
    ↳ SequencesOptions<ulong> { UseCompression = true,
    ↳ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
799 using (var scope2 = new TempLinksTestScope(useSequences: true))
800 {
801     scope1.Links.UseUnicode();
802     scope2.Links.UseUnicode();
803
804     var compressor1 = scope1.Sequences;
805     var compressor2 = scope2.Sequences;
806
807     var compressed1 = new ulong[arrays.Length];
808     var compressed2 = new ulong[arrays.Length];
809
810     var sw1 = Stopwatch.StartNew();
811
812     var START = 0;
813     var END = arrays.Length;
814
815     for (int i = START; i < END; i++)
816     {
817         compressed1[i] = compressor1.Create(arrays[i]);
818     }
819
820     var elapsed1 = sw1.Elapsed;
821
822     var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
823
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($"Compressor: {elapsed1}, Balanced sequence creator:
    ↳ {elapsed2}");
834
835     Assert.True(elapsed1 > elapsed2);
836
837     // Checks
838     for (int i = START; i < END; i++)
839     {
840         var sequence1 = compressed1[i];
841         var sequence2 = compressed2[i];
842
843         if (sequence1 != _constants.Null && sequence2 != _constants.Null)
844         {
845             var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
    ↳ scope1.Links);
846
847             var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
    ↳ scope2.Links);
848
849             Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
850         }
851     }
852
853     Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
854     Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
855
856     Debug.WriteLine($"{{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
    ↳ totalCharacters}} | {{(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
    ↳ totalCharacters}}");
857
858     // Can be worse than balanced variant
859     //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
860
861     //compressor1.ValidateFrequencies();
862 }
863
864
865 [Fact]
866 public static void AllTreeBreakDownAtSequencesCreationBugTest()
867 {
868     // Made out of AllPossibleConnectionsTest test.
869

```

```

870 //const long sequenceLength = 5; //100% bug
871 const long sequenceLength = 4; //100% bug
872 //const long sequenceLength = 3; //100% _no_bug (ok)
873
874 using (var scope = new TempLinksTestScope(useSequences: true))
875 {
876     var links = scope.Links;
877     var sequences = scope.Sequences;
878
879     var sequence = new ulong[sequenceLength];
880     for (var i = 0; i < sequenceLength; i++)
881     {
882         sequence[i] = links.Create();
883     }
884
885     var createResults = sequences.CreateAllVariants2(sequence);
886
887     Global.Trash = createResults;
888
889     for (var i = 0; i < sequenceLength; i++)
890     {
891         links.Delete(sequence[i]);
892     }
893 }
894
895 [Fact]
896 public static void AllPossibleConnectionsTest()
897 {
898     const long sequenceLength = 5;
899
900     using (var scope = new TempLinksTestScope(useSequences: true))
901     {
902         var links = scope.Links;
903         var sequences = scope.Sequences;
904
905         var sequence = new ulong[sequenceLength];
906         for (var i = 0; i < sequenceLength; i++)
907         {
908             sequence[i] = links.Create();
909         }
910
911         var createResults = sequences.CreateAllVariants2(sequence);
912         var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
913
914         for (var i = 0; i < 1; i++)
915         {
916             var sw1 = Stopwatch.StartNew();
917             var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
918
919             var sw2 = Stopwatch.StartNew();
920             var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();
921
922             var sw3 = Stopwatch.StartNew();
923             var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();
924
925             var sw4 = Stopwatch.StartNew();
926             var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();
927
928             Global.Trash = searchResults3;
929             Global.Trash = searchResults4; //-V3008
930
931             var intersection1 = createResults.Intersect(searchResults1).ToList();
932             Assert.True(intersection1.Count == createResults.Length);
933
934             var intersection2 = reverseResults.Intersect(searchResults1).ToList();
935             Assert.True(intersection2.Count == reverseResults.Length);
936
937             var intersection0 = searchResults1.Intersect(searchResults2).ToList();
938             Assert.True(intersection0.Count == searchResults2.Count);
939
940             var intersection3 = searchResults2.Intersect(searchResults3).ToList();
941             Assert.True(intersection3.Count == searchResults3.Count);
942
943             var intersection4 = searchResults3.Intersect(searchResults4).ToList();
944             Assert.True(intersection4.Count == searchResults4.Count);
945         }
946
947         for (var i = 0; i < sequenceLength; i++)
948         {
949

```



```

950         links.Delete(sequence[i]);
951     }
952 }
953 }
954
955 [Fact(Skip = "Correct implementation is pending")]
956 public static void CalculateAllUsagesTest()
957 {
958     const long sequenceLength = 3;
959
960     using (var scope = new TempLinksTestScope(useSequences: true))
961     {
962         var links = scope.Links;
963         var sequences = scope.Sequences;
964
965         var sequence = new ulong[sequenceLength];
966         for (var i = 0; i < sequenceLength; i++)
967         {
968             sequence[i] = links.Create();
969         }
970
971         var createResults = sequences.CreateAllVariants2(sequence);
972
973         //var reverseResults =
974         ↪ sequences.CreateAllVariants2(sequence.Reverse().ToArray());
975
976         for (var i = 0; i < 1; i++)
977         {
978             var linksTotalUsages1 = new ulong[links.Count() + 1];
979
980             sequences.CalculateAllUsages(linksTotalUsages1);
981
982             var linksTotalUsages2 = new ulong[links.Count() + 1];
983
984             sequences.CalculateAllUsages2(linksTotalUsages2);
985
986             var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
987             Assert.True(intersection1.Count == linksTotalUsages2.Length);
988         }
989
990         for (var i = 0; i < sequenceLength; i++)
991         {
992             links.Delete(sequence[i]);
993         }
994     }
995 }
996 }

```

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

```

1  using System.IO;
2  using Platform.Disposables;
3  using Platform.Data.Doublets.ResizableDirectMemory;
4  using Platform.Data.Doublets.Sequences;
5  using Platform.Data.Doublets.Decorators;
6
7  namespace Platform.Data.Doublets.Tests
8  {
9      public class TempLinksTestScope : DisposableBase
10     {
11         public readonly ILinks<ulong> MemoryAdapter;
12         public readonly SynchronizedLinks<ulong> Links;
13         public readonly Sequences.Sequences Sequences;
14         public readonly string TempFilename;
15         public readonly string TempTransactionLogFilename;
16         private readonly bool _deleteFiles;
17
18         public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool
19         ↪ useLog = false)
20             : this(new SequencesOptions<ulong>(), deleteFiles, useSequences, useLog)
21         {
22         }
23
24         public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles =
25         ↪ true, bool useSequences = false, bool useLog = false)
26         {
27             _deleteFiles = deleteFiles;
28             TempFilename = Path.GetTempFileName();
29             TempTransactionLogFilename = Path.GetTempFileName();

```

```

29     var coreMemoryAdapter = new UInt64ResizableDirectMemoryLinks(TempFilename);
30
31     MemoryAdapter = useLog ? (ILinks<ulong>)new
        ↳ UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
        ↳ coreMemoryAdapter;
32
33     Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
34     if (useSequences)
35     {
36         Sequences = new Sequences.Sequences(Links, sequencesOptions);
37     }
38 }
39
40 protected override void Dispose(bool manual, bool wasDisposed)
41 {
42     if (!wasDisposed)
43     {
44         Links.Unsync.DisposeIfPossible();
45         if (_deleteFiles)
46         {
47             DeleteFiles();
48         }
49     }
50 }
51
52 public void DeleteFiles()
53 {
54     File.Delete(TempFilename);
55     File.Delete(TempTransactionLogFilename);
56 }
57 }
58 }

```

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

```

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

```

```

47     Assert.True(equalityComparer.Equals(link.Source, linkAddress));
48     Assert.True(equalityComparer.Equals(link.Target, linkAddress));
49
50     // Update link to reference null (prepare for delete)
51     var updated = links.Update(linkAddress, constants.Null, constants.Null);
52
53     Assert.True(equalityComparer.Equals(updated, linkAddress));
54
55     link = new Link<T>(links.GetLink(linkAddress));
56
57     Assert.True(equalityComparer.Equals(link.Source, constants.Null));
58     Assert.True(equalityComparer.Equals(link.Target, constants.Null));
59
60     // Delete link
61     links.Delete(linkAddress);
62
63     Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));
64
65     setter = new Setter<T>(constants.Null);
66     links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
67
68     Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
69 }
70
71 public static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
72 {
73     // Constants
74     var constants = links.Constants;
75     var equalityComparer = EqualityComparer<T>.Default;
76
77     var h106E = new Hybrid<T>(106L, isExternal: true);
78     var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
79     var h108E = new Hybrid<T>(-108L);
80
81     Assert.Equal(106L, h106E.AbsoluteValue);
82     Assert.Equal(107L, h107E.AbsoluteValue);
83     Assert.Equal(108L, h108E.AbsoluteValue);
84
85     // Create Link (External -> External)
86     var linkAddress1 = links.Create();
87
88     links.Update(linkAddress1, h106E, h108E);
89
90     var link1 = new Link<T>(links.GetLink(linkAddress1));
91
92     Assert.True(equalityComparer.Equals(link1.Source, h106E));
93     Assert.True(equalityComparer.Equals(link1.Target, h108E));
94
95     // Create Link (Internal -> External)
96     var linkAddress2 = links.Create();
97
98     links.Update(linkAddress2, linkAddress1, h108E);
99
100     var link2 = new Link<T>(links.GetLink(linkAddress2));
101
102     Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
103     Assert.True(equalityComparer.Equals(link2.Target, h108E));
104
105     // Create Link (Internal -> Internal)
106     var linkAddress3 = links.Create();
107
108     links.Update(linkAddress3, linkAddress1, linkAddress2);
109
110     var link3 = new Link<T>(links.GetLink(linkAddress3));
111
112     Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
113     Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));
114
115     // Search for created link
116     var setter1 = new Setter<T>(constants.Null);
117     links.Each(h106E, h108E, setter1.SetAndReturnFalse);
118
119     Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));
120
121     // Search for nonexistent link
122     var setter2 = new Setter<T>(constants.Null);
123     links.Each(h106E, h107E, setter2.SetAndReturnFalse);
124
125     Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));
126

```

```

127 // Update link to reference null (prepare for delete)
128 var updated = links.Update(linkAddress3, constants.Null, constants.Null);
129
130 Assert.True(equalityComparer.Equals(updated, linkAddress3));
131
132 link3 = new Link<T>(links.GetLink(linkAddress3));
133
134 Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
135 Assert.True(equalityComparer.Equals(link3.Target, constants.Null));
136
137 // Delete link
138 links.Delete(linkAddress3);
139
140 Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Two));
141
142 var setter3 = new Setter<T>(constants.Null);
143 links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);
144
145 Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
146 }
147
148 public static void TestMultipleRandomCreationsAndDeletions<TLink>(this ILinks<TLink>
149     ↪ 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         Assert.True(created == (Integer<TLink>)links.Count());
180         for (var i = 0; i < N; i++)
181         {
182             TLink link = (Integer<TLink>)(i + 1);
183             if (links.Exists(link))
184             {
185                 links.Delete(link);
186                 deleted++;
187             }
188         }
189         Assert.True((Integer<TLink>)links.Count() == 0);
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;

```



```

89 public static void TransactionAutoRevertedTest()
90 {
91     // Auto Reverted (Because no commit at transaction)
92     using (var scope = new TempLinksTestScope(useLog: true))
93     {
94         var links = scope.Links;
95         var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
96         using (var transaction = transactionsLayer.BeginTransaction())
97         {
98             var l1 = links.Create();
99             var l2 = links.Create();
100
101             links.Update(l2, l2, l1, l2);
102         }
103
104         Assert.Equal(0UL, links.Count());
105
106         links.Unsync.DisposeIfPossible();
107
108         var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope.TempTransactionLogFilename);
109         Assert.Single(transitions);
110     }
111 }
112
113 [Fact]
114 public static void TransactionUserCodeErrorNoDataSavedTest()
115 {
116     // User Code Error (Autoreverted), no data saved
117     var itself = _constants.Itself;
118
119     TempLinksTestScope lastScope = null;
120     try
121     {
122         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
123             ↪ useLog: true))
124         {
125             var links = scope.Links;
126             var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecorator)
127             ↪ atorBase<ulong>)links.Unsync).Links;
128             using (var transaction = transactionsLayer.BeginTransaction())
129             {
130                 var l1 = links.CreateAndUpdate(itself, itself);
131                 var l2 = links.CreateAndUpdate(itself, itself);
132
133                 l2 = links.Update(l2, l2, l1, l2);
134
135                 links.CreateAndUpdate(l2, itself);
136                 links.CreateAndUpdate(l2, itself);
137
138                 //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope.TempTransactionLogFilename);
139                 ↪ tion>(scope.TempTransactionLogFilename);
140
141                 l2 = links.Update(l2, l1);
142
143                 links.Delete(l2);
144
145                 ExceptionThrower();
146
147                 transaction.Commit();
148             }
149
150             Global.Trash = links.Count();
151         }
152     }
153     catch
154     {
155         Assert.False(lastScope == null);
156
157         var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(lastScope.TempTransactionLogFilename);
158         ↪ astScope.TempTransactionLogFilename);
159
160         Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
161             ↪ transitions[0].After.IsNull());
162
163         lastScope.DeleteFiles();
164     }
165 }

```

```

162 [Fact]
163 public static void TransactionUserCodeErrorSomeDataSavedTest()
164 {
165     // User Code Error (Autoreverted), some data saved
166     var itself = _constants.Itself;
167
168     TempLinksTestScope lastScope = null;
169     try
170     {
171         ulong l1;
172         ulong l2;
173
174         using (var scope = new TempLinksTestScope(useLog: true))
175         {
176             var links = scope.Links;
177             l1 = links.CreateAndUpdate(itself, itself);
178             l2 = links.CreateAndUpdate(itself, itself);
179
180             l2 = links.Update(l2, l2, l1, l2);
181
182             links.CreateAndUpdate(l2, itself);
183             links.CreateAndUpdate(l2, itself);
184
185             links.Unsync.DisposeIfPossible();
186
187             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
188                 ↪ scope.TempTransactionLogFilename);
189         }
190
191         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
192             ↪ useLog: true))
193         {
194             var links = scope.Links;
195             var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
196             using (var transaction = transactionsLayer.BeginTransaction())
197             {
198                 l2 = links.Update(l2, l1);
199
200                 links.Delete(l2);
201
202                 ExceptionThrower();
203
204                 transaction.Commit();
205             }
206
207             Global.Trash = links.Count();
208         }
209     }
210     catch
211     {
212         Assert.False(lastScope == null);
213
214         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
215             ↪ Scope.TempTransactionLogFilename);
216
217         lastScope.DeleteFiles();
218     }
219 }
220
221 [Fact]
222 public static void TransactionCommit()
223 {
224     var itself = _constants.Itself;
225
226     var tempDatabaseFilename = Path.GetTempFileName();
227     var tempTransactionLogFilename = Path.GetTempFileName();
228
229     // Commit
230     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
231         ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
232         ↪ tempTransactionLogFilename))
233     using (var links = new UInt64Links(memoryAdapter))
234     {
235         using (var transaction = memoryAdapter.BeginTransaction())
236         {
237             var l1 = links.CreateAndUpdate(itself, itself);
238             var l2 = links.CreateAndUpdate(itself, itself);
239
240             Global.Trash = links.Update(l2, l2, l1, l2);
241
242         }
243     }
244 }

```

```

236         links.Delete(l1);
237
238         transaction.Commit();
239     }
240
241     Global.Trash = links.Count();
242 }
243
244 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
↪ sactionLogFilename);
245
246 }
247
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
↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
↪ tempTransactionLogFilename))
258     using (var links = new UInt64Links(memoryAdapter))
259     {
260         using (var transaction = memoryAdapter.BeginTransaction())
261         {
262             var l1 = links.CreateAndUpdate(itself, itself);
263             var l2 = links.CreateAndUpdate(itself, itself);
264
265             Global.Trash = links.Update(l2, l2, l1, l2);
266
267             links.Delete(l1);
268
269             transaction.Commit();
270         }
271
272         Global.Trash = links.Count();
273     }
274
275     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
↪ sactionLogFilename);
276
277     // Damage database
278
279     FileHelpers.WriteFirst(tempTransactionLogFilename, new
↪ UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
280
281     // Try load damaged database
282     try
283     {
284         // TODO: Fix
285         using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
↪ tempTransactionLogFilename))
286         using (var links = new UInt64Links(memoryAdapter))
287         {
288             Global.Trash = links.Count();
289         }
290     }
291     catch (NotSupportedException ex)
292     {
293         Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
↪ yet.");
294     }
295
296     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
↪ sactionLogFilename);
297
298     File.Delete(tempDatabaseFilename);
299     File.Delete(tempTransactionLogFilename);
300 }
301
302 [Fact]
303 public static void Bug1Test()
304 {

```



```

305     var tempDatabaseFilename = Path.GetTempFileName();
306     var tempTransactionLogFilename = Path.GetTempFileName();
307
308     var itself = _constants.Itself;
309
310     // User Code Error (Autoreverted), some data saved
311     try
312     {
313         ulong l1;
314         ulong l2;
315
316         using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
317             ↳ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
318             ↳ tempTransactionLogFilename))
319         using (var links = new UInt64Links(memoryAdapter))
320         {
321             l1 = links.CreateAndUpdate(itself, itself);
322             l2 = links.CreateAndUpdate(itself, itself);
323
324             l2 = links.Update(l2, l2, l1, l2);
325
326             links.CreateAndUpdate(l2, itself);
327             links.CreateAndUpdate(l2, itself);
328         }
329
330         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
331             ↳ TransactionLogFilename);
332
333         using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
334             ↳ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
335             ↳ tempTransactionLogFilename))
336         using (var links = new UInt64Links(memoryAdapter))
337         {
338             using (var transaction = memoryAdapter.BeginTransaction())
339             {
340                 l2 = links.Update(l2, l1);
341
342                 links.Delete(l2);
343
344                 ExceptionThrower();
345
346                 transaction.Commit();
347             }
348
349             Global.Trash = links.Count();
350         }
351     }
352     catch
353     {
354         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
355             ↳ TransactionLogFilename);
356     }
357
358     File.Delete(tempDatabaseFilename);
359     File.Delete(tempTransactionLogFilename);
360 }
361
362 private static void ExceptionThrower()
363 {
364     throw new Exception();
365 }
366
367 [Fact]
368 public static void PathsTest()
369 {
370     var source = _constants.SourcePart;
371     var target = _constants.TargetPart;
372
373     using (var scope = new TempLinksTestScope())
374     {
375         var links = scope.Links;
376         var l1 = links.CreatePoint();
377         var l2 = links.CreatePoint();
378
379         var r1 = links.GetByKey(l1, source, target, source);
380         var r2 = links.CheckPathExistence(l2, l2, l2, l2);
381     }
382 }

```

```

378 [Fact]
379 public static void RecursiveStringFormattingTest()
380 {
381     using (var scope = new TempLinksTestScope(useSequences: true))
382     {
383         var links = scope.Links;
384         var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.
385
386         var a = links.CreatePoint();
387         var b = links.CreatePoint();
388         var c = links.CreatePoint();
389
390         var ab = links.CreateAndUpdate(a, b);
391         var cb = links.CreateAndUpdate(c, b);
392         var ac = links.CreateAndUpdate(a, c);
393
394         a = links.Update(a, c, b);
395         b = links.Update(b, a, c);
396         c = links.Update(c, a, b);
397
398         Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
399         Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
400         Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
401
402         Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
403             ↳ "(5:(4:5 (6:5 4)) 6)");
404         Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
405             ↳ "(6:(5:(4:5 6) 6) 4)");
406         Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
407             ↳ "(4:(5:4 (6:5 4)) 6)");
408
409         // TODO: Think how to build balanced syntax tree while formatting structure (eg.
410         ↳ "(4:(5:4 6) (6:5 4))" instead of "(4:(5:4 (6:5 4)) 6)"
411
412         Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
413             ↳ "{{5}{5}{4}{6}}");
414         Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
415             ↳ "{{5}{6}{6}{4}}");
416         Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
417             ↳ "{{4}{5}{4}{6}}");
418     }
419 }
420
421 private static void DefaultFormatter(StringBuilder sb, ulong link)
422 {
423     sb.Append(link.ToString());
424 }
425
426 #endregion
427
428 #region Performance
429
430 /*
431 public static void RunAllPerformanceTests()
432 {
433     try
434     {
435         links.TestLinksInSteps();
436     }
437     catch (Exception ex)
438     {
439         ex.WriteToConsole();
440     }
441
442     return;
443
444     try
445     {
446         //ThreadPool.SetMaxThreads(2, 2);
447
448         // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
449         ↳ результат
450         // Также это дополнительно помогает в отладке
451         // Увеличивает вероятность попадания информации в кэши
452         for (var i = 0; i < 10; i++)
453         {
454             //0 - 10 ГБ
455             //Каждые 100 МБ срез цифр

```

```

449         //links.TestGetSourceFunction();
450         //links.TestGetSourceFunctionInParallel();
451         //links.TestGetTargetFunction();
452         //links.TestGetTargetFunctionInParallel();
453         links.Create64BillionLinks();
454
455         links.TestRandomSearchFixed();
456         //links.Create64BillionLinksInParallel();
457         links.TestEachFunction();
458         //links.TestForeach();
459         //links.TestParallelForeach();
460     }
461
462     links.TestDeletionOfAllLinks();
463
464 }
465 catch (Exception ex)
466 {
467     ex.WriteToConsole();
468 }
469 */
470
471 /*
472 public static void TestLinksInSteps()
473 {
474     const long gibibyte = 1024 * 1024 * 1024;
475     const long mebibyte = 1024 * 1024;
476
477     var totalLinksToCreate = gibibyte /
↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
478     var linksStep = 102 * mebibyte /
↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
479
480     var creationMeasurements = new List<TimeSpan>();
481     var searchMeasurements = new List<TimeSpan>();
482     var deletionMeasurements = new List<TimeSpan>();
483
484     GetBaseRandomLoopOverhead(linksStep);
485     GetBaseRandomLoopOverhead(linksStep);
486
487     var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
488
489     ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
490
491     var loops = totalLinksToCreate / linksStep;
492
493     for (int i = 0; i < loops; i++)
494     {
495         creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
496         searchMeasurements.Add(Measure(() => links.RunRandomSearches(linksStep)));
497
498         Console.WriteLine("\rC + S {0}/{1}", i + 1, loops);
499     }
500
501     ConsoleHelpers.Debug();
502
503     for (int i = 0; i < loops; i++)
504     {
505         deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
506
507         Console.WriteLine("\rD {0}/{1}", i + 1, loops);
508     }
509
510     ConsoleHelpers.Debug();
511
512     ConsoleHelpers.Debug("C S D");
513
514     for (int i = 0; i < loops; i++)
515     {
516         ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
↪ searchMeasurements[i], deletionMeasurements[i]);
517     }
518
519     ConsoleHelpers.Debug("C S D (no overhead)");
520
521     for (int i = 0; i < loops; i++)
522     {
523         ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] - stepLoopOverhead,
↪ searchMeasurements[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);

```

```

524     }
525
526     ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
↪ links.Total);
527     }
528
529     private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
↪ amountToCreate)
530     {
531         for (long i = 0; i < amountToCreate; i++)
532             links.Create(0, 0);
533     }
534
535     private static TimeSpan GetBaseRandomLoopOverhead(long loops)
536     {
537         return Measure(() =>
538         {
539             ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
540             ulong result = 0;
541             for (long i = 0; i < loops; i++)
542             {
543                 var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
544                 var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
545
546                 result += maxValue + source + target;
547             }
548             Global.Trash = result;
549         });
550     }
551     */
552
553     [Fact(Skip = "performance test")]
554     public static void GetSourceTest()
555     {
556         using (var scope = new TempLinksTestScope())
557         {
558             var links = scope.Links;
559             ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
↪ Iterations);
560
561             ulong counter = 0;
562
563             //var firstLink = links.First();
564             // Создаём одну связь, из которой будет производить считывание
565             var firstLink = links.Create();
566
567             var sw = Stopwatch.StartNew();
568
569             // Тестируем саму функцию
570             for (ulong i = 0; i < Iterations; i++)
571             {
572                 counter += links.GetSource(firstLink);
573             }
574
575             var elapsedTime = sw.Elapsed;
576
577             var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
578
579             // Удаляем связь, из которой производилось считывание
580             links.Delete(firstLink);
581
582             ConsoleHelpers.Debug(
583                 "{0} Iterations of GetSource function done in {1} ({2} Iterations per
↪ second), counter result: {3}",
584                 Iterations, elapsedTime, (long)iterationsPerSecond, counter);
585         }
586     }
587
588     [Fact(Skip = "performance test")]
589     public static void GetSourceInParallel()
590     {
591         using (var scope = new TempLinksTestScope())
592         {
593             var links = scope.Links;
594             ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
↪ parallel.", Iterations);
595
596             long counter = 0;
597

```

```

598     //var firstLink = links.First();
599     var firstLink = links.Create();
600
601     var sw = Stopwatch.StartNew();
602
603     // Тестируем саму функцию
604     Parallel.For(0, Iterations, x =>
605     {
606         Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
607         //Interlocked.Increment(ref counter);
608     });
609
610     var elapsedTime = sw.Elapsed;
611
612     var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
613
614     links.Delete(firstLink);
615
616     ConsoleHelpers.Debug(
617         "{0} Iterations of GetSource function done in {1} ({2} Iterations per
        ↳ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
618     }
619 }
620
621 [Fact(Skip = "performance test")]
622 public static void TestGetTarget()
623 {
624     using (var scope = new TempLinksTestScope())
625     {
626         var links = scope.Links;
627         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
628             ↳ Iterations);
629
630         ulong counter = 0;
631
632         //var firstLink = links.First();
633         var firstLink = links.Create();
634
635         var sw = Stopwatch.StartNew();
636
637         for (ulong i = 0; i < Iterations; i++)
638         {
639             counter += links.GetTarget(firstLink);
640         }
641
642         var elapsedTime = sw.Elapsed;
643
644         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
645
646         links.Delete(firstLink);
647
648         ConsoleHelpers.Debug(
649             "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
            ↳ second), counter result: {3}",
            Iterations, elapsedTime, (long)iterationsPerSecond, counter);
650     }
651 }
652
653 [Fact(Skip = "performance test")]
654 public static void TestGetTargetInParallel()
655 {
656     using (var scope = new TempLinksTestScope())
657     {
658         var links = scope.Links;
659         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
660             ↳ parallel.", Iterations);
661
662         long counter = 0;
663
664         //var firstLink = links.First();
665         var firstLink = links.Create();
666
667         var sw = Stopwatch.StartNew();
668
669         Parallel.For(0, Iterations, x =>
670         {
671             Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
672             //Interlocked.Increment(ref counter);

```

```

673     });
674
675     var elapsedTime = sw.Elapsed;
676
677     var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
678
679     links.Delete(firstLink);
680
681     ConsoleHelpers.Debug(
682         "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
        ↳ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
683     }
684 }
685
686 // TODO: Заполнить базу данных перед тестом
687 /*
688 [Fact]
689 public void TestRandomSearchFixed()
690 {
691     var tempFilename = Path.GetTempFileName();
692
693     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
694 ↳ DefaultLinksSizeStep))
695     {
696         long iterations = 64 * 1024 * 1024 /
697 ↳ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
698
699         ulong counter = 0;
700         var maxLink = links.Total;
701
702         ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);
703
704         var sw = Stopwatch.StartNew();
705
706         for (var i = iterations; i > 0; i--)
707         {
708             var source =
709 ↳ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
710             var target =
711 ↳ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
712
713             counter += links.Search(source, target);
714         }
715
716         var elapsedTime = sw.Elapsed;
717
718         var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
719
720         ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
        ↳ Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
        ↳ counter);
721     }
722
723     File.Delete(tempFilename);
724 }*/
725
726 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
727 public static void TestRandomSearchAll()
728 {
729     using (var scope = new TempLinksTestScope())
730     {
731         var links = scope.Links;
732         ulong counter = 0;
733
734         var maxLink = links.Count();
735
736         var iterations = links.Count();
737
738         ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
739 ↳ links.Count());
740
741         var sw = Stopwatch.StartNew();
742
743         for (var i = iterations; i > 0; i--)
744         {
745             var linksAddressRange = new Range<ulong>(_constants.MinPossibleIndex,
746 ↳ maxLink);

```

```

743         var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
744         var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
745
746         counter += links.SearchOrDefault(source, target);
747     }
748
749     var elapsedTime = sw.Elapsed;
750
751     var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
752
753     ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
754     ↪ Iterations per second), c: {3}",
755         iterations, elapsedTime, (long)iterationsPerSecond, counter);
756 }
757
758 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
759 public static void TestEach()
760 {
761     using (var scope = new TempLinksTestScope())
762     {
763         var links = scope.Links;
764
765         var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
766
767         ConsoleHelpers.Debug("Testing Each function.");
768
769         var sw = Stopwatch.StartNew();
770
771         links.Each(counter.IncrementAndReturnTrue);
772
773         var elapsedTime = sw.Elapsed;
774
775         var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
776
777         ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
778         ↪ links per second)",
779             counter, elapsedTime, (long)linksPerSecond);
780     }
781 }
782
783 /*
784 [Fact]
785 public static void TestForeach()
786 {
787     var tempFilename = Path.GetTempFileName();
788
789     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
790     ↪ DefaultLinksSizeStep))
791     {
792         ulong counter = 0;
793
794         ConsoleHelpers.Debug("Testing foreach through links.");
795
796         var sw = Stopwatch.StartNew();
797
798         //foreach (var link in links)
799         //{
800             counter++;
801         //}
802
803         var elapsedTime = sw.Elapsed;
804
805         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
806
807         ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
808         ↪ links per second)", counter, elapsedTime, (long)linksPerSecond);
809     }
810
811     File.Delete(tempFilename);
812 }
813 */
814
815 /*
816 [Fact]
817 public static void TestParallelForeach()
818 {
819     var tempFilename = Path.GetTempFileName();

```

```

818         using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪ DefaultLinksSizeStep))
819         {
820             long counter = 0;
821
822             ConsoleHelpers.Debug("Testing parallel foreach through links.");
823
824             var sw = Stopwatch.StartNew();
825
826             //Parallel.ForEach((IEnumerable<ulong>)links, x =>
827             //{
828             //    Interlocked.Increment(ref counter);
829             //});
830
831             var elapsedTime = sw.Elapsed;
832
833             var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
834
835             ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
↪ {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
836         }
837
838         File.Delete(tempFilename);
839     }
840     */
841
842     [Fact(Skip = "performance test")]
843     public static void Create64BillionLinks()
844     {
845         using (var scope = new TempLinksTestScope())
846         {
847             var links = scope.Links;
848             var linksBeforeTest = links.Count();
849
850             long linksToCreate = 64 * 1024 * 1024 /
↪ UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
851
852             ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
853
854             var elapsedTime = Performance.Measure(() =>
855             {
856                 for (long i = 0; i < linksToCreate; i++)
857                 {
858                     links.Create();
859                 }
860             });
861
862             var linksCreated = links.Count() - linksBeforeTest;
863             var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
864
865             ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
866
867             ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
↪ linksCreated, elapsedTime,
868                 (long)linksPerSecond);
869         }
870     }
871
872     [Fact(Skip = "performance test")]
873     public static void Create64BillionLinksInParallel()
874     {
875         using (var scope = new TempLinksTestScope())
876         {
877             var links = scope.Links;
878             var linksBeforeTest = links.Count();
879
880             var sw = Stopwatch.StartNew();
881
882             long linksToCreate = 64 * 1024 * 1024 /
↪ UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
883
884             ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);
885
886             Parallel.For(0, linksToCreate, x => links.Create());
887
888             var elapsedTime = sw.Elapsed;
889
890             var linksCreated = links.Count() - linksBeforeTest;
891             var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
892

```



```

893
894     ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
      ↪ linksCreated, elapsedTime,
895         (long)linksPerSecond);
896 }
897 }
898
899 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
900 public static void TestDeletionOfAllLinks()
901 {
902     using (var scope = new TempLinksTestScope())
903     {
904         var links = scope.Links;
905         var linksBeforeTest = links.Count();
906
907         ConsoleHelpers.Debug("Deleting all links");
908
909         var elapsedTime = Performance.Measure(links.DeleteAll);
910
911         var linksDeleted = linksBeforeTest - links.Count();
912         var linksPerSecond = linksDeleted / elapsedTime.TotalSeconds;
913
914         ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
      ↪ linksDeleted, elapsedTime,
915             (long)linksPerSecond);
916     }
917 }
918
919 #endregion
920 }
921 }

```

./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
      ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, one);
19                 var toUnaryNumberConverter = new AddressToUnaryNumberConverter<ulong>(links,
      ↪ powerOf2ToUnaryNumberConverter);
20                 var random = new System.Random(0);
21                 ulong[] numbers = new ulong[N];
22                 ulong[] unaryNumbers = new ulong[N];
23                 for (int i = 0; i < N; i++)
24                 {
25                     numbers[i] = random.NextUInt64();
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
29                     ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, one);
30                 var addressToUnaryNumberConverter = new
31                     ↪ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
32                 var unaryNumberToAddressConverter = new
33                     ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links,
34                     ↪ powerOf2ToUnaryNumberConverter);
35                 TestCharAndUnicodeSymbolConverters(links, meaningRoot,
36                     ↪ addressToUnaryNumberConverter, unaryNumberToAddressConverter);
37             }
38         }
39
40         [Fact]
41         public static void CharAndRawNumberUnicodeSymbolConvertersTest()
42         {
43             using (var scope = new Scope<Types<HeapResizableDirectMemory,
44                 ↪ ResizableDirectMemoryLinks<ulong>>>())
45             {
46                 var links = scope.Use<ILinks<ulong>>>();
47                 var meaningRoot = links.CreatePoint();
48                 var addressToRawNumberConverter = new AddressToRawNumberConverter<ulong>();
49                 var rawNumberToAddressConverter = new RawNumberToAddressConverter<ulong>();
50                 TestCharAndUnicodeSymbolConverters(links, meaningRoot,
51                     ↪ addressToRawNumberConverter, rawNumberToAddressConverter);
52             }
53         }
54
55         private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
56             ↪ meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
57             ↪ numberToAddressConverter)
58         {
59             var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
60             var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links,
61                 ↪ addressToNumberConverter, unicodeSymbolMarker);
62             var originalCharacter = 'H';
63             var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);
64             var unicodeSymbolCriterionMatcher = new UnicodeSymbolCriterionMatcher<ulong>(links,
65                 ↪ unicodeSymbolMarker);
66             var unicodeSymbolToCharConverter = new UnicodeSymbolToCharConverter<ulong>(links,
67                 ↪ numberToAddressConverter, unicodeSymbolCriterionMatcher);
68             var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
69             Assert.Equal(originalCharacter, resultingCharacter);
70         }
71
72         [Fact]
73         public static void StringAndUnicodeSequenceConvertersTest()
74         {
75             using (var scope = new TempLinksTestScope())
76             {
77                 var links = scope.Links;
```

```

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
77     ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
78     var addressToUnaryNumberConverter = new
79     ↪ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
80     var charToUnicodeSymbolConverter = new
81     ↪ CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
82     ↪ unicodeSymbolMarker);
83
84     var unaryNumberToAddressConverter = new
85     ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links,
86     ↪ powerOf2ToUnaryNumberConverter);
87     var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
88     var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
89     ↪ frequencyMarker, unaryOne, unaryNumberIncrementer);
90     var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
91     ↪ frequencyPropertyMarker, frequencyMarker);
92     var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
93     ↪ frequencyPropertyOperator, frequencyIncrementer);
94     var linkToItsFrequencyNumberConverter = new
95     ↪ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
96     ↪ unaryNumberToAddressConverter);
97     var sequenceToItsLocalElementLevelsConverter = new
98     ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
99     ↪ linkToItsFrequencyNumberConverter);
100    var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
101    ↪ sequenceToItsLocalElementLevelsConverter);
102
103    var stringToUnicodeSymbolConverter = new
104    ↪ StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
105    ↪ index, optimalVariantConverter, unicodeSequenceMarker);
106
107    var originalString = "Hello";
108
109    var unicodeSequenceLink = stringToUnicodeSymbolConverter.Convert(originalString);
110
111    var unicodeSymbolCriterionMatcher = new
112    ↪ UnicodeSymbolCriterionMatcher<ulong>(links, unicodeSymbolMarker);
113    var unicodeSymbolToCharConverter = new
114    ↪ UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
115    ↪ unicodeSymbolCriterionMatcher);
116
117    var unicodeSequenceCriterionMatcher = new
118    ↪ UnicodeSequenceCriterionMatcher<ulong>(links, unicodeSequenceMarker);
119
120    var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
121    ↪ unicodeSymbolCriterionMatcher.IsMatched);
122
123    var unicodeSequenceToStringConverter = new
124    ↪ UnicodeSequenceToStringConverter<ulong>(links,
125    ↪ unicodeSequenceCriterionMatcher, sequenceWalker,
126    ↪ unicodeSymbolToCharConverter);
127
128    var resultingString =
129    ↪ unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
130
131    Assert.Equal(originalString, resultingString);
132
133    }
134
135    }
136
137    }

```

Index

./Platform.Data.Doublets.Tests/ComparisonTests.cs, 140
./Platform.Data.Doublets.Tests/EqualityTests.cs, 141
./Platform.Data.Doublets.Tests/GenericLinksTests.cs, 142
./Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs, 143
./Platform.Data.Doublets.Tests/ReadSequenceTests.cs, 144
./Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs, 145
./Platform.Data.Doublets.Tests/ScopeTests.cs, 146
./Platform.Data.Doublets.Tests/SequencesTests.cs, 146
./Platform.Data.Doublets.Tests/TempLinksTestScope.cs, 161
./Platform.Data.Doublets.Tests/TestExtensions.cs, 162
./Platform.Data.Doublets.Tests/UInt64LinksTests.cs, 164
./Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs, 177
./Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs, 177
./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, 3
./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, 6
./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, 25
./Platform.Data.Doublets/Incrementers/FrequencyIncrementer.cs, 24
./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, 28
./Platform.Data.Doublets/Numbers/Raw/RawNumberToAddressConverter.cs, 28
./Platform.Data.Doublets/Numbers/Unary/AddressToUnaryNumberConverter.cs, 29
./Platform.Data.Doublets/Numbers/Unary/LinkToltsFrequencyNumberConveter.cs, 29
./Platform.Data.Doublets/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs, 30
./Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressAddOperationConverter.cs, 30
./Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressOrOperationConverter.cs, 31
./Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs, 32
./Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs, 33
./Platform.Data.Doublets/ResizableDirectMemory/ResizableDirectMemoryLinks.ListMethods.cs, 42
./Platform.Data.Doublets/ResizableDirectMemory/ResizableDirectMemoryLinks.TreeMethods.cs, 43
./Platform.Data.Doublets/ResizableDirectMemory/ResizableDirectMemoryLinks.cs, 33
./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.ListMethods.cs, 56
./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.TreeMethods.cs, 57
./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.cs, 49
./Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.cs, 63
./Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs, 64
./Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs, 67
./Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs, 67
./Platform.Data.Doublets/Sequences/Converters/SequenceToltsLocalElementLevelsConverter.cs, 69
./Platform.Data.Doublets/Sequences/CriteriaMatchers/DefaultSequenceElementCriterionMatcher.cs, 69
./Platform.Data.Doublets/Sequences/CriteriaMatchers/MarkedSequenceCriterionMatcher.cs, 69
./Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs, 70
./Platform.Data.Doublets/Sequences/DuplicateSegmentsCounter.cs, 70
./Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs, 71
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 73
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs, 75
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToltsFrequencyValueConverter.cs, 75

./Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 75
./Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 76
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 76
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 77
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 77
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 77
./Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 78
./Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 79
./Platform.Data.Doublets/Sequences/HeightProviders/ISequenceHeightProvider.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/Sequences.Experiments.cs, 92
./Platform.Data.Doublets/Sequences/Sequences.cs, 82
./Platform.Data.Doublets/Sequences/SequencesExtensions.cs, 118
./Platform.Data.Doublets/Sequences/SequencesOptions.cs, 118
./Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.cs, 120
./Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs, 120
./Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs, 120
./Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs, 122
./Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs, 122
./Platform.Data.Doublets/Stacks/Stack.cs, 123
./Platform.Data.Doublets/Stacks/StackExtensions.cs, 124
./Platform.Data.Doublets/SynchronizedLinks.cs, 124
./Platform.Data.Doublets/UInt64Link.cs, 125
./Platform.Data.Doublets/UInt64LinkExtensions.cs, 127
./Platform.Data.Doublets/UInt64LinksExtensions.cs, 127
./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs, 130
./Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs, 135
./Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs, 135
./Platform.Data.Doublets/Unicode/UnicodeMap.cs, 136
./Platform.Data.Doublets/Unicode/UnicodeSequenceCriterionMatcher.cs, 138
./Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs, 138
./Platform.Data.Doublets/Unicode/UnicodeSymbolCriterionMatcher.cs, 139
./Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs, 139