./Platform.Data.Doublets/Decorators/LinksCascadeUniquenessAndUsagesResolver.cs

```csharp
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
       ↪   newLinkAddress)
10         {
11             Links.MergeUsages(oldLinkAddress, newLinkAddress);
12             return base.ResolveAddressChangeConflict(oldLinkAddress, newLinkAddress);
13         }
14     }
15 }
```

./Platform.Data.Doublets/Decorators/LinksCascadeUsagesResolver.cs

```csharp
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             this.DeleteAllUsages(linkIndex);
16             Links.Delete(linkIndex);
17         }
18     }
19 }
```

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

```csharp
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         protected LinksDecoratorBase(ILinks<TLink> links) : base(links) => Constants =
       ↪   links.Constants;
13         public virtual TLink Count(IList<TLink> restriction) => Links.Count(restriction);
14         public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
       ↪   => Links.Each(handler, restrictions);
15         public virtual TLink Create() => Links.Create();
16         public virtual TLink Update(IList<TLink> restrictions) => Links.Update(restrictions);
17         public virtual void Delete(TLink link) => Links.Delete(link);
18     }
19 }
```

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

```csharp
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)
   ↪  => Links.Each(handler, restrictions);
25
26        public virtual TLink Create() => Links.Create();
27
28        public virtual TLink Update(IList<TLink> restrictions) => Links.Update(restrictions);
29
30        public virtual void Delete(TLink link) => Links.Delete(link);
31
32        protected override bool AllowMultipleDisposeCalls => true;
33
34        protected override void Dispose(bool manual, bool wasDisposed)
35        {
36            if (!wasDisposed)
37            {
38                Links.DisposeIfPossible();
39            }
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
   ↪  be external (hybrid link's raw number).
9      public class LinksInnerReferenceExistenceValidator<TLink> : LinksDecoratorBase<TLink>
10     {
11         public LinksInnerReferenceExistenceValidator(ILinks<TLink> links) : base(links) { }
12
13         public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
14         {
15             Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
16             return Links.Each(handler, restrictions);
17         }
18
19         public override TLink Update(IList<TLink> restrictions)
20         {
21             // TODO: Possible values: null, ExistentLink or NonExistentHybrid(ExternalReference)
22             Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
23             return Links.Update(restrictions);
24         }
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 =
   ↪  EqualityComparer<TLink>.Default;
11
12         public LinksItselfConstantToSelfReferenceResolver(ILinks<TLink> links) : base(links) { }
13
14         public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
```

```
15          {
16              var constants = Constants;
17              var itselfConstant = constants.Itself;
18              var indexPartConstant = constants.IndexPart;
19              var sourcePartConstant = constants.SourcePart;
20              var targetPartConstant = constants.TargetPart;
21              var restrictionsCount = restrictions.Count;
22              if (!_equalityComparer.Equals(constants.Any, itselfConstant)
23               && (((restrictionsCount > indexPartConstant) &&
                 ↪  _equalityComparer.Equals(restrictions[indexPartConstant], itselfConstant))
24                  || ((restrictionsCount > sourcePartConstant) &&
                 ↪  _equalityComparer.Equals(restrictions[sourcePartConstant], itselfConstant))
25                  || ((restrictionsCount > targetPartConstant) &&
                 ↪  _equalityComparer.Equals(restrictions[targetPartConstant], itselfConstant))))
26              {
27                  // Itself constant is not supported for Each method right now, skipping execution
28                  return constants.Continue;
29              }
30              return Links.Each(handler, restrictions);
31          }
32
33          public override TLink Update(IList<TLink> restrictions) =>
             ↪  Links.Update(Links.ResolveConstantAsSelfReference(Constants.Itself, restrictions));
34      }
35  }
```

./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
            ↪  location inside ResizableDirectMemoryLinks.
10      /// This in turn will require to implement not a list of empty links, but a list of ranges
            ↪  to store it more efficiently.
11      /// </remarks>
12      public class LinksNonExistentDependenciesCreator<TLink> : LinksDecoratorBase<TLink>
13      {
14          public LinksNonExistentDependenciesCreator(ILinks<TLink> links) : base(links) { }
15
16          public override TLink Update(IList<TLink> restrictions)
17          {
18              var constants = Constants;
19              Links.EnsureCreated(restrictions[constants.SourcePart],
                 ↪  restrictions[constants.TargetPart]);
20              return Links.Update(restrictions);
21          }
22      }
23  }
```

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

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

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

```csharp
using System.Collections.Generic;

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

namespace Platform.Data.Doublets.Decorators
{
    public class LinksUniquenessResolver<TLink> : LinksDecoratorBase<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
          EqualityComparer<TLink>.Default;

        public LinksUniquenessResolver(ILinks<TLink> links) : base(links) { }

        public override TLink Update(IList<TLink> restrictions)
        {
            var newLinkAddress = Links.SearchOrDefault(restrictions[Constants.SourcePart],
              restrictions[Constants.TargetPart]);
            if (_equalityComparer.Equals(newLinkAddress, default))
            {
                return Links.Update(restrictions);
            }
            return ResolveAddressChangeConflict(restrictions[Constants.IndexPart],
              newLinkAddress);
        }

        protected virtual TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
          newLinkAddress)
        {
            if (!_equalityComparer.Equals(oldLinkAddress, newLinkAddress) &&
              Links.Exists(oldLinkAddress))
            {
                Delete(oldLinkAddress);
            }
            return newLinkAddress;
        }
    }
}
```

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

```csharp
using System.Collections.Generic;

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

namespace Platform.Data.Doublets.Decorators
{
    public class LinksUniquenessValidator<TLink> : LinksDecoratorBase<TLink>
    {
        public LinksUniquenessValidator(ILinks<TLink> links) : base(links) { }

        public override TLink Update(IList<TLink> restrictions)
        {
            Links.EnsureDoesNotExists(restrictions[Constants.SourcePart],
              restrictions[Constants.TargetPart]);
            return Links.Update(restrictions);
        }
    }
}
```

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

```csharp
using System.Collections.Generic;

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

namespace Platform.Data.Doublets.Decorators
{
    public class LinksUsagesValidator<TLink> : LinksDecoratorBase<TLink>
    {
        public LinksUsagesValidator(ILinks<TLink> links) : base(links) { }

        public override TLink Update(IList<TLink> restrictions)
        {
            Links.EnsureNoUsages(restrictions[Constants.IndexPart]);
            return Links.Update(restrictions);
        }

        public override void Delete(TLink link)
        {
            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
       ↪  поддеревьев для AVL баланса и флагов нитей: 2 в степени(64 минус 5 равно 59 ) равно 576
       ↪  460 752 303 423 488
23    /// Желательно реализовать поддержку переключения между деревьями и битовыми индексами
       ↪  (битовыми строками) - вариант матрицы (выстраеваемой лениво).
24    ///
25    /// Решить отключать ли проверки при компиляции под Release. Т.е. исключения будут
       ↪  выбрасываться только при #if DEBUG
26    /// </remarks>
27    public class UInt64Links : LinksDisposableDecoratorBase<ulong>
28    {
29        public UInt64Links(ILinks<ulong> links) : base(links) { }
30
31        public override ulong Each(Func<IList<ulong>, ulong> handler, IList<ulong> restrictions)
32        {
33            this.EnsureLinkIsAnyOrExists(restrictions);
34            return Links.Each(handler, restrictions);
35        }
36
37        public override ulong Create() => 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
               ↪  support
48            if (restrictions.Count != 3)
49            {
50                throw new NotSupportedException();
51            }
```

```
52          var indexPartConstant = constants.IndexPart;
53          var updatedLink = restrictions[indexPartConstant];
54          this.EnsureLinkExists(updatedLink,
   ↪       $"{nameof(restrictions)}[{nameof(indexPartConstant)}]");
55          var sourcePartConstant = constants.SourcePart;
56          var newSource = restrictions[sourcePartConstant];
57          this.EnsureLinkIsItselfOrExists(newSource,
   ↪       $"{nameof(restrictions)}[{nameof(sourcePartConstant)}]");
58          var targetPartConstant = constants.TargetPart;
59          var newTarget = restrictions[targetPartConstant];
60          this.EnsureLinkIsItselfOrExists(newTarget,
   ↪       $"{nameof(restrictions)}[{nameof(targetPartConstant)}]");
61          var existedLink = nullConstant;
62          var itselfConstant = constants.Itself;
63          if (newSource != itselfConstant && newTarget != itselfConstant)
64          {
65              existedLink = this.SearchOrDefault(newSource, newTarget);
66          }
67          if (existedLink == nullConstant)
68          {
69              var before = Links.GetLink(updatedLink);
70              if (before[sourcePartConstant] != newSource || before[targetPartConstant] !=
   ↪           newTarget)
71              {
72                  Links.Update(updatedLink, newSource == itselfConstant ? updatedLink :
   ↪               newSource,
73                                              newTarget == itselfConstant ? updatedLink :
   ↪                                           newTarget);
74              }
75              return updatedLink;
76          }
77          else
78          {
79              return this.MergeAndDelete(updatedLink, existedLink);
80          }
81      }
82
83      public override void Delete(ulong linkIndex)
84      {
85          Links.EnsureLinkExists(linkIndex);
86          Links.EnforceResetValues(linkIndex);
87          this.DeleteAllUsages(linkIndex);
88          Links.Delete(linkIndex);
89      }
90   }
91 }
```

./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
   ↪   by itself. But can cause creation (update from nothing) or deletion (update to nothing).
16     ///
17     /// TODO: Decide to change to IDoubletLinks or not to change. (Better to create
   ↪   DefaultUniLinksBase, that contains logic itself and can be implemented using both
   ↪   IDoubletLinks and ILinks.)
18     /// </remarks>
19     internal class UniLinks<TLink> : LinksDecoratorBase<TLink>, IUniLinks<TLink>
20     {
21         private static readonly EqualityComparer<TLink> _equalityComparer =
   ↪       EqualityComparer<TLink>.Default;
22
23         public UniLinks(ILinks<TLink> links) : base(links) { }
24
25         private struct Transition
26         {
27             public IList<TLink> Before;
```

```csharp
        public IList<TLink> After;

        public Transition(IList<TLink> before, IList<TLink> after)
        {
            Before = before;
            After = after;
        }
    }

    //public static readonly TLink NullConstant = Use<LinksCombinedConstants<TLink, TLink,
    //  int>>.Single.Null;
    //public static readonly IReadOnlyList<TLink> NullLink = new
    //  ReadOnlyCollection<TLink>(new List<TLink> { NullConstant, NullConstant, NullConstant
    //  });

    // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution
    //  (Links-Expression)
    public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>, TLink>
    //  matchedHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
    //  substitutedHandler)
    {
        ////List<Transition> transitions = null;
        ////if (!restriction.IsNullOrEmpty())
        ////{
        ////    // Есть причина делать проход (чтение)
        ////    if (matchedHandler != null)
        ////    {
        ////        if (!substitution.IsNullOrEmpty())
        ////        {
        ////            // restriction => { 0, 0, 0 } | { 0 } // Create
        ////            // substitution => { itself, 0, 0 } | { itself, itself, itself } //
        //  Create / Update
        ////            // substitution => { 0, 0, 0 } | { 0 } // Delete
        ////            transitions = new List<Transition>();
        ////            if (Equals(substitution[Constants.IndexPart], Constants.Null))
        ////            {
        ////                // If index is Null, that means we always ignore every other
        //  value (they are also Null by definition)
        ////                var matchDecision = matchedHandler(, NullLink);
        ////                if (Equals(matchDecision, Constants.Break))
        ////                    return false;
        ////                if (!Equals(matchDecision, Constants.Skip))
        ////                    transitions.Add(new Transition(matchedLink, newValue));
        ////            }
        ////            else
        ////            {
        ////                Func<T, bool> handler;
        ////                handler = link =>
        ////                {
        ////                    var matchedLink = Memory.GetLinkValue(link);
        ////                    var newValue = Memory.GetLinkValue(link);
        ////                    newValue[Constants.IndexPart] = Constants.Itself;
        ////                    newValue[Constants.SourcePart] =
        //  Equals(substitution[Constants.SourcePart], Constants.Itself) ?
        //  matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
        ////                    newValue[Constants.TargetPart] =
        //  Equals(substitution[Constants.TargetPart], Constants.Itself) ?
        //  matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
        ////                    var matchDecision = matchedHandler(matchedLink, newValue);
        ////                    if (Equals(matchDecision, Constants.Break))
        ////                        return false;
        ////                    if (!Equals(matchDecision, Constants.Skip))
        ////                        transitions.Add(new Transition(matchedLink, newValue));
        ////                    return true;
        ////                };
        ////                if (!Memory.Each(handler, restriction))
        ////                    return Constants.Break;
        ////            }
        ////        }
        ////        else
        ////        {
        ////            Func<T, bool> handler = link =>
        ////            {
        ////                var matchedLink = Memory.GetLinkValue(link);
        ////                var matchDecision = matchedHandler(matchedLink, matchedLink);
        ////                return !Equals(matchDecision, Constants.Break);
        ////            };
```

```
 93    ////               if (!Memory.Each(handler, restriction))
 94    ////                   return Constants.Break;
 95    ////           }
 96    ////       }
 97    ////       else
 98    ////       {
 99    ////           if (substitution != null)
100    ////           {
101    ////               transitions = new List<IList<T>>();
102    ////               Func<T, bool> handler = link =>
103    ////               {
104    ////                   var matchedLink = Memory.GetLinkValue(link);
105    ////                   transitions.Add(matchedLink);
106    ////                   return true;
107    ////               };
108    ////               if (!Memory.Each(handler, restriction))
109    ////                   return Constants.Break;
110    ////           }
111    ////           else
112    ////           {
113    ////               return Constants.Continue;
114    ////           }
115    ////       }
116    ////}
117    ////if (substitution != null)
118    ////{
119    ////    // Есть причина делать замену (запись)
120    ////    if (substitutedHandler != null)
121    ////    {
122    ////    }
123    ////    else
124    ////    {
125    ////    }
126    ////}
127    ////return Constants.Continue;
128
129    //if (restriction.IsNullOrEmpty()) // Create
130    //{
131    //    substitution[Constants.IndexPart] = Memory.AllocateLink();
132    //    Memory.SetLinkValue(substitution);
133    //}
134    //else if (substitution.IsNullOrEmpty()) // Delete
135    //{
136    //    Memory.FreeLink(restriction[Constants.IndexPart]);
137    //}
138    //else if (restriction.EqualTo(substitution)) // Read or ("repeat" the state) // Each
139    //{
140    //    // No need to collect links to list
141    //    // Skip == Continue
142    //    // No need to check substituedHandler
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
        ↪   substitution handling.
175     //                  var substitutedDecision = substitutedHandler(matchedLink,
        ↪   newValue);
176     //                  if (Equals(substitutedDecision, Constants.Break))
177     //                      return Constants.Break;
178     //                  if (Equals(substitutedDecision, Constants.Continue))
179     //                  {
180     //                      // Actual update here
181     //                      Memory.SetLinkValue(newValue);
182     //                  }
183     //                  if (Equals(substitutedDecision, Constants.Skip))
184     //                  {
185     //                      // Cancel the update. TODO: decide use separate Cancel
        ↪   constant or Skip is enough?
186     //                  }
187     //              }
188     //          }
189     //      }
190     //}
191         return Constants.Continue;
192     }
193
194     public TLink Trigger(IList<TLink> patternOrCondition, Func<IList<TLink>, TLink>
        ↪   matchHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
        ↪   substitutionHandler)
195     {
196         if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
197         {
198             return Constants.Continue;
199         }
200         else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO:
        ↪   Check if it is a correct condition
201         {
202             // Or it only applies to trigger without matchHandler.
203             throw new NotImplementedException();
204         }
205         else if (!substitution.IsNullOrEmpty()) // Creation
206         {
207             var before = ArrayPool<TLink>.Empty;
208             // Что должно означать False здесь? Остановиться (перестать идти) или пропустить
        ↪   (пройти мимо) или пустить (взять)?
209             if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
        ↪   Constants.Break))
210             {
211                 return Constants.Break;
212             }
213             var after = (IList<TLink>)substitution.ToArray();
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)
```

```csharp
                {
                    var linkToDelete = patternOrCondition[0];
                    var before = Links.GetLink(linkToDelete);
                    if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                        Constants.Break))
                    {
                        return Constants.Break;
                    }
                    var after = ArrayPool<TLink>.Empty;
                    Links.Update(linkToDelete, Constants.Null, Constants.Null);
                    Links.Delete(linkToDelete);
                    if (matchHandler != null)
                    {
                        return substitutionHandler(before, after);
                    }
                    return Constants.Continue;
                }
                else
                {
                    throw new NotSupportedException();
                }
            }
            else // Replace / Update
            {
                if (patternOrCondition.Count == 1) //-V3125
                {
                    var linkToUpdate = patternOrCondition[0];
                    var before = Links.GetLink(linkToUpdate);
                    if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                        Constants.Break))
                    {
                        return Constants.Break;
                    }
                    var after = (IList<TLink>)substitution.ToArray(); //-V3125
                    if (_equalityComparer.Equals(after[0], default))
                    {
                        after[0] = linkToUpdate;
                    }
                    if (substitution.Count == 1)
                    {
                        if (!_equalityComparer.Equals(substitution[0], linkToUpdate))
                        {
                            after = Links.GetLink(substitution[0]);
                            Links.Update(linkToUpdate, Constants.Null, Constants.Null);
                            Links.Delete(linkToUpdate);
                        }
                    }
                    else if (substitution.Count == 3)
                    {
                        Links.Update(after);
                    }
                    else
                    {
                        throw new NotSupportedException();
                    }
                    if (matchHandler != null)
                    {
                        return substitutionHandler(before, after);
                    }
                    return Constants.Continue;
                }
                else
                {
                    throw new NotSupportedException();
                }
            }
        }

        /// <remarks>
        /// IList[IList[IList[T]]]
        /// |       |       |     |||
        /// |       |       ------ ||
        /// |       |        link  ||
        /// |       ------------- |
        /// |           change    |
        ///  -------------------
        ///        changes
        /// </remarks>
```

```csharp
316         public IList<IList<IList<TLink>>> Trigger(IList<TLink> condition, IList<TLink>
        ↪  substitution)
317         {
318             var changes = new List<IList<IList<TLink>>>();
319             Trigger(condition, AlwaysContinue, substitution, (before, after) =>
320             {
321                 var change = new[] { before, after };
322                 changes.Add(change);
323                 return Constants.Continue;
324             });
325             return changes;
326         }
327
328         private TLink AlwaysContinue(IList<TLink> linkToMatch) => Constants.Continue;
329     }
330 }
```

./Platform.Data.Doublets/DoubletComparer.cs

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

./Platform.Data.Doublets/Doublet.cs

```csharp
1  using System;
2  using System.Collections.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets
7  {
8      public struct Doublet<T> : IEquatable<Doublet<T>>
9      {
10         private static readonly EqualityComparer<T> _equalityComparer =
        ↪  EqualityComparer<T>.Default;
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)
        ↪  && _equalityComparer.Equals(Target, other.Target);
24
25         public override bool Equals(object obj) => obj is Doublet<T> doublet ?
        ↪  base.Equals(doublet) : false;
26
27         public override int GetHashCode() => (Source, Target).GetHashCode();
28     }
29 }
```

./Platform.Data.Doublets/Hybrid.cs

```csharp
1  using System;
2  using System.Reflection;
3  using Platform.Reflection;
```

```csharp
using Platform.Converters;
using Platform.Exceptions;

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

namespace Platform.Data.Doublets
{
    public class Hybrid<T>
    {
        public readonly T Value;
        public bool IsNothing => Convert.ToInt64(To.Signed(Value)) == 0;
        public bool IsInternal => Convert.ToInt64(To.Signed(Value)) > 0;
        public bool IsExternal => Convert.ToInt64(To.Signed(Value)) < 0;
        public long AbsoluteValue =>
            Platform.Numbers.Math.Abs(Convert.ToInt64(To.Signed(Value)));

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

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

        public Hybrid(object value, bool isExternal)
        {
            var signedType = Type<T>.SignedVersion;
            var signedValue = Convert.ChangeType(value, signedType);
            var abs = typeof(Platform.Numbers.Math).GetTypeInfo().GetMethod("Abs").MakeGenericMe
                thod(signedType);
            var negate = typeof(Platform.Numbers.Math).GetTypeInfo().GetMethod("Negate").MakeGen
                ericMethod(signedType);
            var absoluteValue = abs.Invoke(null, new[] { signedValue });
            var resultValue = isExternal ? negate.Invoke(null, new[] { absoluteValue }) :
                absoluteValue;
            Value = To.UnsignedAs<T>(resultValue);
        }

        public static implicit operator Hybrid<T>(T integer) => new Hybrid<T>(integer);

        public static explicit operator Hybrid<T>(ulong integer) => new Hybrid<T>(integer);

        public static explicit operator Hybrid<T>(long integer) => new Hybrid<T>(integer);

        public static explicit operator Hybrid<T>(uint integer) => new Hybrid<T>(integer);

        public static explicit operator Hybrid<T>(int integer) => new Hybrid<T>(integer);

        public static explicit operator Hybrid<T>(ushort integer) => new Hybrid<T>(integer);

        public static explicit operator Hybrid<T>(short integer) => new Hybrid<T>(integer);

        public static explicit operator Hybrid<T>(byte integer) => new Hybrid<T>(integer);

        public static explicit operator Hybrid<T>(sbyte integer) => new Hybrid<T>(integer);

        public static implicit operator T(Hybrid<T> hybrid) => hybrid.Value;

        public static explicit operator ulong(Hybrid<T> hybrid) =>
            Convert.ToUInt64(hybrid.Value);

        public static explicit operator long(Hybrid<T> hybrid) => hybrid.AbsoluteValue;

        public static explicit operator uint(Hybrid<T> hybrid) => Convert.ToUInt32(hybrid.Value);

        public static explicit operator int(Hybrid<T> hybrid) =>
            Convert.ToInt32(hybrid.AbsoluteValue);

        public static explicit operator ushort(Hybrid<T> hybrid) =>
            Convert.ToUInt16(hybrid.Value);

        public static explicit operator short(Hybrid<T> hybrid) =>
            Convert.ToInt16(hybrid.AbsoluteValue);

        public static explicit operator byte(Hybrid<T> hybrid) => Convert.ToByte(hybrid.Value);

        public static explicit operator sbyte(Hybrid<T> hybrid) =>
            Convert.ToSByte(hybrid.AbsoluteValue);
```

```csharp
        public override string ToString() => IsNothing ? default(T) == null ? "Nothing" :
            default(T).ToString() : IsExternal ? $"<{AbsoluteValue}>" : Value.ToString();
    }
}
```

./Platform.Data.Doublets/ILinks.cs

```csharp
using Platform.Data.Constants;

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

namespace Platform.Data.Doublets
{
    public interface ILinks<TLink> : ILinks<TLink, LinksCombinedConstants<TLink, TLink, int>>
    {
    }
}
```

./Platform.Data.Doublets/ILinksExtensions.cs

```csharp
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.CompilerServices;
using Platform.Ranges;
using Platform.Collections.Arrays;
using Platform.Numbers;
using Platform.Random;
using Platform.Setters;
using Platform.Data.Exceptions;

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

namespace Platform.Data.Doublets
{
    public static class ILinksExtensions
    {
        public static void RunRandomCreations<TLink>(this ILinks<TLink> links, long
            amountOfCreations)
        {
            for (long i = 0; i < amountOfCreations; i++)
            {
                var linksAddressRange = new Range<ulong>(0, (Integer<TLink>)links.Count());
                Integer<TLink> source = RandomHelpers.Default.NextUInt64(linksAddressRange);
                Integer<TLink> target = RandomHelpers.Default.NextUInt64(linksAddressRange);
                links.CreateAndUpdate(source, target);
            }
        }

        public static void RunRandomSearches<TLink>(this ILinks<TLink> links, long
            amountOfSearches)
        {
            for (long i = 0; i < amountOfSearches; i++)
            {
                var linkAddressRange = new Range<ulong>(1, (Integer<TLink>)links.Count());
                Integer<TLink> source = RandomHelpers.Default.NextUInt64(linkAddressRange);
                Integer<TLink> target = RandomHelpers.Default.NextUInt64(linkAddressRange);
                links.SearchOrDefault(source, target);
            }
        }

        public static void RunRandomDeletions<TLink>(this ILinks<TLink> links, long
            amountOfDeletions)
        {
            var min = (ulong)amountOfDeletions > (Integer<TLink>)links.Count() ? 1 :
                (Integer<TLink>)links.Count() - (ulong)amountOfDeletions;
            for (long i = 0; i < amountOfDeletions; i++)
            {
                var linksAddressRange = new Range<ulong>(min, (Integer<TLink>)links.Count());
                Integer<TLink> link = RandomHelpers.Default.NextUInt64(linksAddressRange);
                links.Delete(link);
                if ((Integer<TLink>)links.Count() < min)
                {
                    break;
                }
            }
        }

        /// <remarks>
```

```csharp
        /// TODO: Возможно есть очень простой способ это сделать.
        /// (Например просто удалить файл, или изменить его размер таким образом,
        /// чтобы удалился весь контент)
        /// Например через _header->AllocatedLinks в ResizableDirectMemoryLinks
        /// </remarks>
        public static void DeleteAll<TLink>(this ILinks<TLink> links)
        {
            var equalityComparer = EqualityComparer<TLink>.Default;
            var comparer = Comparer<TLink>.Default;
            for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
            ↪  Arithmetic.Decrement(i))
            {
                links.Delete(i);
                if (!equalityComparer.Equals(links.Count(), Arithmetic.Decrement(i)))
                {
                    i = links.Count();
                }
            }
        }

        public static TLink First<TLink>(this ILinks<TLink> links)
        {
            TLink firstLink = default;
            var equalityComparer = EqualityComparer<TLink>.Default;
            if (equalityComparer.Equals(links.Count(), default))
            {
                throw new Exception("В хранилище нет связей.");
            }
            links.Each(links.Constants.Any, links.Constants.Any, link =>
            {
                firstLink = link[links.Constants.IndexPart];
                return links.Constants.Break;
            });
            if (equalityComparer.Equals(firstLink, default))
            {
                throw new Exception("В процессе поиска по хранилищу не было найдено связей.");
            }
            return firstLink;
        }

        public static bool IsInnerReference<TLink>(this ILinks<TLink> links, TLink reference)
        {
            var constants = links.Constants;
            var comparer = Comparer<TLink>.Default;
            return comparer.Compare(constants.MinPossibleIndex, reference) >= 0 &&
            ↪  comparer.Compare(reference, constants.MaxPossibleIndex) <= 0;
        }

        #region Paths

        /// <remarks>
        /// TODO: Как так? Как то что ниже может быть корректно?
        /// Скорее всего практически не применимо
        /// Предполагалось, что можно было конвертировать формируемый в проходе через
        ↪  SequenceWalker
        /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
        /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
        /// </remarks>
        public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[]
        ↪  path)
        {
            var current = path[0];
            //EnsureLinkExists(current, "path");
            if (!links.Exists(current))
            {
                return false;
            }
            var equalityComparer = EqualityComparer<TLink>.Default;
            var constants = links.Constants;
            for (var i = 1; i < path.Length; i++)
            {
                var next = path[i];
                var values = links.GetLink(current);
                var source = values[constants.SourcePart];
                var target = values[constants.TargetPart];
                if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
                ↪  next))
                {
```

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

```csharp
198        /// Возвращает индекс начальной (Source) связи для указанной связи.
199        /// </summary>
200        /// <param name="links">Хранилище связей.</param>
201        /// <param name="link">Связь представленная списком, состоящим из её адреса и
   ↪   содержимого.</param>
202        /// <returns>Индекс начальной связи для указанной связи.</returns>
203        [MethodImpl(MethodImplOptions.AggressiveInlining)]
204        public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
   ↪   link[links.Constants.SourcePart];
205
206        /// <summary>
207        /// Возвращает индекс конечной (Target) связи для указанной связи.
208        /// </summary>
209        /// <param name="links">Хранилище связей.</param>
210        /// <param name="link">Индекс связи.</param>
211        /// <returns>Индекс конечной связи для указанной связи.</returns>
212        [MethodImpl(MethodImplOptions.AggressiveInlining)]
213        public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
   ↪   links.GetLink(link)[links.Constants.TargetPart];
214
215        /// <summary>
216        /// Возвращает индекс конечной (Target) связи для указанной связи.
217        /// </summary>
218        /// <param name="links">Хранилище связей.</param>
219        /// <param name="link">Связь представленная списком, состоящим из её адреса и
   ↪   содержимого.</param>
220        /// <returns>Индекс конечной связи для указанной связи.</returns>
221        [MethodImpl(MethodImplOptions.AggressiveInlining)]
222        public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
   ↪   link[links.Constants.TargetPart];
223
224        /// <summary>
225        /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
   ↪   (handler) для каждой подходящей связи.
226        /// </summary>
227        /// <param name="links">Хранилище связей.</param>
228        /// <param name="handler">Обработчик каждой подходящей связи.</param>
229        /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
   ↪   может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
   ↪   Any - отсутствие ограничения, 1..∞ конкретный адрес связи.</param>
230        /// <returns>True, в случае если проход по связям не был прерван и False в обратном
   ↪   случае.</returns>
231        [MethodImpl(MethodImplOptions.AggressiveInlining)]
232        public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
   ↪   handler, params TLink[] restrictions)
233            => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
   ↪   links.Constants.Continue);
234
235        /// <summary>
236        /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
   ↪   (handler) для каждой подходящей связи.
237        /// </summary>
238        /// <param name="links">Хранилище связей.</param>
239        /// <param name="source">Значение, определяющее соответствующие шаблону связи.
   ↪   (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
   ↪   Constants.Any - любое начало, 1..∞ конкретное начало)</param>
240        /// <param name="target">Значение, определяющее соответствующие шаблону связи.
   ↪   (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
   ↪   Constants.Any - любой конец, 1..∞ конкретный конец)</param>
241        /// <param name="handler">Обработчик каждой подходящей связи.</param>
242        /// <returns>True, в случае если проход по связям не был прерван и False в обратном
   ↪   случае.</returns>
243        [MethodImpl(MethodImplOptions.AggressiveInlining)]
244        public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
   ↪   Func<TLink, bool> handler)
245        {
246            var constants = links.Constants;
247            return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
   ↪   constants.Break, constants.Any, source, target);
248        }
249
250        /// <summary>
251        /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
   ↪   (handler) для каждой подходящей связи.
252        /// </summary>
253        /// <param name="links">Хранилище связей.</param>
```

```csharp
254         /// <param name="source">Значение, определяющее соответствующие шаблону связи.
        ↪   (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
        ↪   Constants.Any - любое начало, 1..∞ конкретное начало)</param>
255         /// <param name="target">Значение, определяющее соответствующие шаблону связи.
        ↪   (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
        ↪   Constants.Any - любой конец, 1..∞ конкретный конец)</param>
256         /// <param name="handler">Обработчик каждой подходящей связи.</param>
257         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
        ↪   случае.</returns>
258         [MethodImpl(MethodImplOptions.AggressiveInlining)]
259         public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
        ↪   Func<IList<TLink>, TLink> handler)
260         {
261             var constants = links.Constants;
262             return links.Each(handler, constants.Any, source, target);
263         }
264
265         [MethodImpl(MethodImplOptions.AggressiveInlining)]
266         public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
        ↪   restrictions)
267         {
268             long arraySize = (Integer<TLink>)links.Count(restrictions);
269             var array = new IList<TLink>[arraySize];
270             if (arraySize > 0)
271             {
272                 var filler = new ArrayFiller<IList<TLink>, TLink>(array,
                ↪   links.Constants.Continue);
273                 links.Each(filler.AddAndReturnConstant, restrictions);
274             }
275             return array;
276         }
277
278         [MethodImpl(MethodImplOptions.AggressiveInlining)]
279         public static IList<TLink> AllIndices<TLink>(this ILinks<TLink> links, params TLink[]
        ↪   restrictions)
280         {
281             long arraySize = (Integer<TLink>)links.Count(restrictions);
282             var array = new TLink[arraySize];
283             if (arraySize > 0)
284             {
285                 var filler = new ArrayFiller<TLink, TLink>(array, links.Constants.Continue);
286                 links.Each(filler.AddFirstAndReturnConstant, restrictions);
287             }
288             return array;
289         }
290
291         /// <summary>
292         /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
        ↪   в хранилище связей.
293         /// </summary>
294         /// <param name="links">Хранилище связей.</param>
295         /// <param name="source">Начало связи.</param>
296         /// <param name="target">Конец связи.</param>
297         /// <returns>Значение, определяющее существует ли связь.</returns>
298         [MethodImpl(MethodImplOptions.AggressiveInlining)]
299         public static bool Exists<TLink>(this ILinks<TLink> links, TLink source, TLink target)
        ↪   => Comparer<TLink>.Default.Compare(links.Count(links.Constants.Any, source, target),
        ↪   default) > 0;
300
301         #region Ensure
302         // TODO: May be move to EnsureExtensions or make it both there and here
303
304         [MethodImpl(MethodImplOptions.AggressiveInlining)]
305         public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
        ↪   reference, string argumentName)
306         {
307             if (links.IsInnerReference(reference) && !links.Exists(reference))
308             {
309                 throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
310             }
311         }
312
313         [MethodImpl(MethodImplOptions.AggressiveInlining)]
314         public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
        ↪   IList<TLink> restrictions, string argumentName)
315         {
316             for (int i = 0; i < restrictions.Count; i++)
```

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

```
386            var createdLinks = new List<TLink>();
387            var equalityComparer = EqualityComparer<TLink>.Default;
388            TLink createdLink = creator();
389            while (!equalityComparer.Equals(createdLink, (Integer<TLink>)max))
390            {
391                createdLinks.Add(createdLink);
392            }
393            for (var i = 0; i < createdLinks.Count; i++)
394            {
395                if (!nonExistentAddresses.Contains((Integer<TLink>)createdLinks[i]))
396                {
397                    links.Delete(createdLinks[i]);
398                }
399            }
400        }
401    }

402

403    #endregion

404

405    /// <param name="links">Хранилище связей.</param>
406    public static ulong CountUsages<TLink>(this ILinks<TLink> links, TLink link)
407    {
408        var constants = links.Constants;
409        var values = links.GetLink(link);
410        ulong usagesAsSource = (Integer<TLink>)links.Count(new Link<TLink>(constants.Any,
           ↪  link, constants.Any));
411        var equalityComparer = EqualityComparer<TLink>.Default;
412        if (equalityComparer.Equals(values[constants.SourcePart], link))
413        {
414            usagesAsSource--;
415        }
416        ulong usagesAsTarget = (Integer<TLink>)links.Count(new Link<TLink>(constants.Any,
           ↪  constants.Any, link));
417        if (equalityComparer.Equals(values[constants.TargetPart], link))
418        {
419            usagesAsTarget--;
420        }
421        return usagesAsSource + usagesAsTarget;
422    }

423

424    /// <param name="links">Хранилище связей.</param>
425    [MethodImpl(MethodImplOptions.AggressiveInlining)]
426    public static bool HasUsages<TLink>(this ILinks<TLink> links, TLink link) =>
       ↪  links.CountUsages(link) > 0;

427

428    /// <param name="links">Хранилище связей.</param>
429    [MethodImpl(MethodImplOptions.AggressiveInlining)]
430    public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source,
       ↪  TLink target)
431    {
432        var constants = links.Constants;
433        var values = links.GetLink(link);
434        var equalityComparer = EqualityComparer<TLink>.Default;
435        return equalityComparer.Equals(values[constants.SourcePart], source) &&
           ↪  equalityComparer.Equals(values[constants.TargetPart], target);
436    }

437

438    /// <summary>
439    /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
440    /// </summary>
441    /// <param name="links">Хранилище связей.</param>
442    /// <param name="source">Индекс связи, которая является началом для искомой
       ↪  связи.</param>
443    /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
444    /// <returns>Индекс искомой связи с указанными Source (началом) и Target
       ↪  (концом).</returns>
445    [MethodImpl(MethodImplOptions.AggressiveInlining)]
446    public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
       ↪  target)
447    {
448        var contants = links.Constants;
449        var setter = new Setter<TLink, TLink>(contants.Continue, contants.Break, default);
450        links.Each(setter.SetFirstAndReturnFalse, contants.Any, source, target);
451        return setter.Result;
452    }

453

454    /// <param name="links">Хранилище связей.</param>
455    [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
{
    var link = links.Create();
    return links.Update(link, link, link);
}

/// <param name="links">Хранилище связей.</param>
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
    target) => links.Update(links.Create(), source, target);

/// <summary>
/// Обновляет связь с указанными началом (Source) и концом (Target)
/// на связь с указанными началом (NewSource) и концом (NewTarget).
/// </summary>
/// <param name="links">Хранилище связей.</param>
/// <param name="link">Индекс обновляемой связи.</param>
/// <param name="newSource">Индекс связи, которая является началом связи, на которую
///     выполняется обновление.</param>
/// <param name="newTarget">Индекс связи, которая является концом связи, на которую
///     выполняется обновление.</param>
/// <returns>Индекс обновлённой связи.</returns>
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static TLink Update<TLink>(this ILinks<TLink> links, TLink link, TLink newSource,
    TLink newTarget) => links.Update(new Link<TLink>(link, newSource, newTarget));

/// <summary>
/// Обновляет связь с указанными началом (Source) и концом (Target)
/// на связь с указанными началом (NewSource) и концом (NewTarget).
/// </summary>
/// <param name="links">Хранилище связей.</param>
/// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
///     может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
///     Itself - требование установить ссылку на себя, 1..∞ конкретный адрес другой
///     связи.</param>
/// <returns>Индекс обновлённой связи.</returns>
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
{
    if (restrictions.Length == 2)
    {
        return links.MergeAndDelete(restrictions[0], restrictions[1]);
    }
    if (restrictions.Length == 4)
    {
        return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
            restrictions[2], restrictions[3]);
    }
    else
    {
        return links.Update(restrictions);
    }
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
    links, TLink constant, IList<TLink> restrictions)
{
    var equalityComparer = EqualityComparer<TLink>.Default;
    var constants = links.Constants;
    var index = restrictions[constants.IndexPart];
    var source = restrictions[constants.SourcePart];
    var target = restrictions[constants.TargetPart];
    source = equalityComparer.Equals(source, constant) ? index : source;
    target = equalityComparer.Equals(target, constant) ? index : target;
    return new Link<TLink>(index, source, target);
}

/// <summary>
/// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
///     с указанными Source (началом) и Target (концом).
/// </summary>
/// <param name="links">Хранилище связей.</param>
/// <param name="source">Индекс связи, которая является началом на создаваемой
///     связи.</param>
/// <param name="target">Индекс связи, которая является концом для создаваемой
///     связи.</param>
```

```csharp
        /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
        ↪ target)
        {
            var link = links.SearchOrDefault(source, target);
            if (EqualityComparer<TLink>.Default.Equals(link, default))
            {
                link = links.CreateAndUpdate(source, target);
            }
            return link;
        }

        /// <summary>
        /// Обновляет связь с указанными началом (Source) и концом (Target)
        /// на связь с указанными началом (NewSource) и концом (NewTarget).
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="source">Индекс связи, которая является началом обновляемой
        ↪ связи.</param>
        /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
        /// <param name="newSource">Индекс связи, которая является началом связи, на которую
        ↪ выполняется обновление.</param>
        /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
        ↪ выполняется обновление.</param>
        /// <returns>Индекс обновлённой связи.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
        ↪ TLink target, TLink newSource, TLink newTarget)
        {
            var equalityComparer = EqualityComparer<TLink>.Default;
            var link = links.SearchOrDefault(source, target);
            if (equalityComparer.Equals(link, default))
            {
                return links.CreateAndUpdate(newSource, newTarget);
            }
            if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
            ↪ target))
            {
                return link;
            }
            return links.Update(link, newSource, newTarget);
        }

        /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
        /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
        ↪ target)
        {
            var link = links.SearchOrDefault(source, target);
            if (!EqualityComparer<TLink>.Default.Equals(link, default))
            {
                links.Delete(link);
                return link;
            }
            return default;
        }

        /// <summary>Удаляет несколько связей.</summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="deletedLinks">Список адресов связей к удалению.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
        {
            for (int i = 0; i < deletedLinks.Count; i++)
            {
                links.Delete(deletedLinks[i]);
            }
        }

        /// <remarks>Before execution of this method ensure that deleted link is detached (all
        ↪ values - source and target are reset to null) or it might enter into infinite
        ↪ recursion.</remarks>
        public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
        {
```

```csharp
            var anyConstant = links.Constants.Any;
            var usagesAsSourceQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
            links.DeleteByQuery(usagesAsSourceQuery);
            var usagesAsTargetQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
            links.DeleteByQuery(usagesAsTargetQuery);
        }

        public static void DeleteByQuery<TLink>(this ILinks<TLink> links, Link<TLink> query)
        {
            var count = (Integer<TLink>)links.Count(query);
            if (count > 0)
            {
                var queryResult = new TLink[count];
                var queryResultFiller = new ArrayFiller<TLink, TLink>(queryResult,
                ↪  links.Constants.Continue);
                links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
                for (var i = (long)count - 1; i >= 0; i--)
                {
                    links.Delete(queryResult[i]);
                }
            }
        }

        // TODO: Move to Platform.Data
        public static bool AreValuesReset<TLink>(this ILinks<TLink> links, TLink linkIndex)
        {
            var nullConstant = links.Constants.Null;
            var equalityComparer = EqualityComparer<TLink>.Default;
            var link = links.GetLink(linkIndex);
            for (int i = 1; i < link.Count; i++)
            {
                if (!equalityComparer.Equals(link[i], nullConstant))
                {
                    return false;
                }
            }
            return true;
        }

        // TODO: Create a universal version of this method in Platform.Data (with using of for
        ↪  loop)
        public static void ResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
        {
            var nullConstant = links.Constants.Null;
            var updateRequest = new Link<TLink>(linkIndex, nullConstant, nullConstant);
            links.Update(updateRequest);
        }

        // TODO: Create a universal version of this method in Platform.Data (with using of for
        ↪  loop)
        public static void EnforceResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
        {
            if (!links.AreValuesReset(linkIndex))
            {
                links.ResetValues(linkIndex);
            }
        }

        /// <summary>
        /// Merging two usages graphs, all children of old link moved to be children of new link
        ↪  or deleted.
        /// </summary>
        public static TLink MergeUsages<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
        ↪  TLink newLinkIndex)
        {
            var equalityComparer = EqualityComparer<TLink>.Default;
            if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
            {
                var constants = links.Constants;
                var usagesAsSourceQuery = new Link<TLink>(constants.Any, oldLinkIndex,
                ↪  constants.Any);
                long usagesAsSourceCount = (Integer<TLink>)links.Count(usagesAsSourceQuery);
                var usagesAsTargetQuery = new Link<TLink>(constants.Any, constants.Any,
                ↪  oldLinkIndex);
                long usagesAsTargetCount = (Integer<TLink>)links.Count(usagesAsTargetQuery);
                var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(oldLinkIndex)) &&
                ↪  usagesAsSourceCount == 1 && usagesAsTargetCount == 1;
                if (!isStandalonePoint)
```

```
660                     {
661                         var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
662                         if (totalUsages > 0)
663                         {
664                             var usages = ArrayPool.Allocate<TLink>(totalUsages);
665                             var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
                                ↪ links.Constants.Continue);
666                             var i = 0L;
667                             if (usagesAsSourceCount > 0)
668                             {
669                                 links.Each(usagesFiller.AddFirstAndReturnConstant,
                                    ↪ usagesAsSourceQuery);
670                                 for (; i < usagesAsSourceCount; i++)
671                                 {
672                                     var usage = usages[i];
673                                     if (!equalityComparer.Equals(usage, oldLinkIndex))
674                                     {
675                                         links.Update(usage, newLinkIndex, links.GetTarget(usage));
676                                     }
677                                 }
678                             }
679                             if (usagesAsTargetCount > 0)
680                             {
681                                 links.Each(usagesFiller.AddFirstAndReturnConstant,
                                    ↪ usagesAsTargetQuery);
682                                 for (; i < usages.Length; i++)
683                                 {
684                                     var usage = usages[i];
685                                     if (!equalityComparer.Equals(usage, oldLinkIndex))
686                                     {
687                                         links.Update(usage, links.GetSource(usage), newLinkIndex);
688                                     }
689                                 }
690                             }
691                             ArrayPool.Free(usages);
692                         }
693                     }
694                 }
695             return newLinkIndex;
696         }
697
698         /// <summary>
699         /// Replace one link with another (replaced link is deleted, children are updated or
            ↪ deleted).
700         /// </summary>
701         [MethodImpl(MethodImplOptions.AggressiveInlining)]
702         public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
            ↪ TLink newLinkIndex)
703         {
704             var equalityComparer = EqualityComparer<TLink>.Default;
705             if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
706             {
707                 links.MergeUsages(oldLinkIndex, newLinkIndex);
708                 links.Delete(oldLinkIndex);
709             }
710             return newLinkIndex;
711         }
712     }
713 }
```

./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 =
           ↪ EqualityComparer<TLink>.Default;
11
12         private readonly TLink _frequencyMarker;
13         private readonly TLink _unaryOne;
14         private readonly IIncrementer<TLink> _unaryNumberIncrementer;
15
16         public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
           ↪ IIncrementer<TLink> unaryNumberIncrementer)
```

```
17              : base(links)
18          {
19              _frequencyMarker = frequencyMarker;
20              _unaryOne = unaryOne;
21              _unaryNumberIncrementer = unaryNumberIncrementer;
22          }
23
24          public TLink Increment(TLink frequency)
25          {
26              if (_equalityComparer.Equals(frequency, default))
27              {
28                  return Links.GetOrCreate(_unaryOne, _frequencyMarker);
29              }
30              var source = Links.GetSource(frequency);
31              var incrementedSource = _unaryNumberIncrementer.Increment(source);
32              return Links.GetOrCreate(incrementedSource, _frequencyMarker);
33          }
34      }
35  }
```

./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 =
            ↪ EqualityComparer<TLink>.Default;
11
12          private readonly TLink _unaryOne;
13
14          public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) : base(links) =>
            ↪ _unaryOne = unaryOne;
15
16          public TLink Increment(TLink unaryNumber)
17          {
18              if (_equalityComparer.Equals(unaryNumber, _unaryOne))
19              {
20                  return Links.GetOrCreate(_unaryOne, _unaryOne);
21              }
22              var source = Links.GetSource(unaryNumber);
23              var target = Links.GetTarget(unaryNumber);
24              if (_equalityComparer.Equals(source, target))
25              {
26                  return Links.GetOrCreate(unaryNumber, _unaryOne);
27              }
28              else
29              {
30                  return Links.GetOrCreate(source, Increment(target));
31              }
32          }
33      }
34  }
```

./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>,
        ↪ LinksCombinedConstants<TLink, TLink, int>>, ILinks<TLink>
8       {
9       }
10  }
```

./Platform.Data.Doublets/Link.cs

```
1   using System;
2   using System.Collections;
3   using System.Collections.Generic;
4   using Platform.Exceptions;
5   using Platform.Ranges;
6   using Platform.Singletons;
7   using Platform.Collections.Lists;
```

```csharp
using Platform.Data.Constants;

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

namespace Platform.Data.Doublets
{
    /// <summary>
    /// Структура описывающая уникальную связь.
    /// </summary>
    public struct Link<TLink> : IEquatable<Link<TLink>>, IReadOnlyList<TLink>, IList<TLink>
    {
        public static readonly Link<TLink> Null = new Link<TLink>();

        private static readonly LinksCombinedConstants<bool, TLink, int> _constants =
            Default<LinksCombinedConstants<bool, TLink, int>>.Instance;
        private static readonly EqualityComparer<TLink> _equalityComparer =
            EqualityComparer<TLink>.Default;

        private const int Length = 3;

        public readonly TLink Index;
        public readonly TLink Source;
        public readonly TLink Target;

        public Link(params TLink[] values)
        {
            Index = values.Length > _constants.IndexPart ? values[_constants.IndexPart] :
                _constants.Null;
            Source = values.Length > _constants.SourcePart ? values[_constants.SourcePart] :
                _constants.Null;
            Target = values.Length > _constants.TargetPart ? values[_constants.TargetPart] :
                _constants.Null;
        }

        public Link(IList<TLink> values)
        {
            Index = values.Count > _constants.IndexPart ? values[_constants.IndexPart] :
                _constants.Null;
            Source = values.Count > _constants.SourcePart ? values[_constants.SourcePart] :
                _constants.Null;
            Target = values.Count > _constants.TargetPart ? values[_constants.TargetPart] :
                _constants.Null;
        }

        public Link(TLink index, TLink source, TLink target)
        {
            Index = index;
            Source = source;
            Target = target;
        }

        public Link(TLink source, TLink target)
            : this(_constants.Null, source, target)
        {
            Source = source;
            Target = target;
        }

        public static Link<TLink> Create(TLink source, TLink target) => new Link<TLink>(source,
            target);

        public override int GetHashCode() => (Index, Source, Target).GetHashCode();

        public bool IsNull() => _equalityComparer.Equals(Index, _constants.Null)
                             && _equalityComparer.Equals(Source, _constants.Null)
                             && _equalityComparer.Equals(Target, _constants.Null);

        public override bool Equals(object other) => other is Link<TLink> &&
            Equals((Link<TLink>)other);

        public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
                                              && _equalityComparer.Equals(Source, other.Source)
                                              && _equalityComparer.Equals(Target, other.Target);

        public static string ToString(TLink index, TLink source, TLink target) => $"({index}:
            {source}->{target})";

        public static string ToString(TLink source, TLink target) => $"({source}->{target})";

        public static implicit operator TLink[](Link<TLink> link) => link.ToArray();
```

```csharp
        public static implicit operator Link<TLink>(TLink[] linkArray) => new
        ↪   Link<TLink>(linkArray);

        public override string ToString() => _equalityComparer.Equals(Index, _constants.Null) ?
        ↪   ToString(Source, Target) : ToString(Index, Source, Target);

        #region IList

        public int Count => Length;

        public bool IsReadOnly => true;

        public TLink this[int index]
        {
            get
            {
                Ensure.Always.ArgumentInRange(index, new Range<int>(0, Length - 1),
                ↪   nameof(index));
                if (index == _constants.IndexPart)
                {
                    return Index;
                }
                if (index == _constants.SourcePart)
                {
                    return Source;
                }
                if (index == _constants.TargetPart)
                {
                    return Target;
                }
                throw new NotSupportedException(); // Impossible path due to
                ↪   Ensure.ArgumentInRange
            }
            set => throw new NotSupportedException();
        }

        IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

        public IEnumerator<TLink> GetEnumerator()
        {
            yield return Index;
            yield return Source;
            yield return Target;
        }

        public void Add(TLink item) => throw new NotSupportedException();

        public void Clear() => throw new NotSupportedException();

        public bool Contains(TLink item) => IndexOf(item) >= 0;

        public void CopyTo(TLink[] array, int arrayIndex)
        {
            Ensure.Always.ArgumentNotNull(array, nameof(array));
            Ensure.Always.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
            ↪   nameof(arrayIndex));
            if (arrayIndex + Length > array.Length)
            {
                throw new InvalidOperationException();
            }
            array[arrayIndex++] = Index;
            array[arrayIndex++] = Source;
            array[arrayIndex] = Target;
        }

        public bool Remove(TLink item) => Throw.A.NotSupportedExceptionAndReturn<bool>();

        public int IndexOf(TLink item)
        {
            if (_equalityComparer.Equals(Index, item))
            {
                return _constants.IndexPart;
            }
            if (_equalityComparer.Equals(Source, item))
            {
                return _constants.SourcePart;
            }
            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) =>
           ↪  Point<TLink>.IsFullPoint(link);
8          public static bool IsPartialPoint<TLink>(this Link<TLink> link) =>
           ↪  Point<TLink>.IsPartialPoint(link);
9      }
10 }
```

./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/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)
```

```
            }
            return target;
        }
    }
}
```

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

```csharp
using System;
using System.Collections.Generic;
using Platform.Interfaces;

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

namespace Platform.Data.Doublets.Numbers.Unary
{
    public class LinkToItsFrequencyNumberConveter<TLink> : LinksOperatorBase<TLink>,
        IConverter<Doublet<TLink>, TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
            EqualityComparer<TLink>.Default;

        private readonly IPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
        private readonly IConverter<TLink> _unaryNumberToAddressConverter;

        public LinkToItsFrequencyNumberConveter(
            ILinks<TLink> links,
            IPropertyOperator<TLink, TLink> frequencyPropertyOperator,
            IConverter<TLink> unaryNumberToAddressConverter)
            : base(links)
        {
            _frequencyPropertyOperator = frequencyPropertyOperator;
            _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
        }

        public TLink Convert(Doublet<TLink> doublet)
        {
            var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
            if (_equalityComparer.Equals(link, default))
            {
                throw new ArgumentException($"Link ({doublet}) not found.", nameof(doublet));
            }
            var frequency = _frequencyPropertyOperator.Get(link);
            if (_equalityComparer.Equals(frequency, default))
            {
                return default;
            }
            var frequencyNumber = Links.GetSource(frequency);
            return _unaryNumberToAddressConverter.Convert(frequencyNumber);
        }
    }
}
```

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

```csharp
using System.Collections.Generic;
using Platform.Exceptions;
using Platform.Interfaces;
using Platform.Ranges;

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

namespace Platform.Data.Doublets.Numbers.Unary
{
    public class PowerOf2ToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
        IConverter<int, TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
            EqualityComparer<TLink>.Default;

        private readonly TLink[] _unaryNumberPowersOf2;

        public PowerOf2ToUnaryNumberConverter(ILinks<TLink> links, TLink one) : base(links)
        {
            _unaryNumberPowersOf2 = new TLink[64];
            _unaryNumberPowersOf2[0] = one;
        }

        public TLink Convert(int power)
        {
            Ensure.Always.ArgumentInRange(power, new Range<int>(0, _unaryNumberPowersOf2.Length
                - 1), nameof(power));
```

```
25                if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
26                {
27                    return _unaryNumberPowersOf2[power];
28                }
29                var previousPowerOf2 = Convert(power - 1);
30                var powerOf2 = Links.GetOrCreate(previousPowerOf2, previousPowerOf2);
31                _unaryNumberPowersOf2[power] = powerOf2;
32                return powerOf2;
33            }
34        }
35    }
```

./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>,
        ↪  IConverter<TLink>
11      {
12          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
13
14          private Dictionary<TLink, TLink> _unaryToUInt64;
15          private readonly TLink _unaryOne;
16
17          public UnaryNumberToAddressAddOperationConverter(ILinks<TLink> links, TLink unaryOne)
18              : base(links)
19          {
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        }

72        [MethodImpl(MethodImplOptions.AggressiveInlining)]
73        private static TLink Double(TLink number) => (Integer<TLink>)((Integer<TLink>)number *
   ↪  2UL);
74    }
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;

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

9  namespace Platform.Data.Doublets.Numbers.Unary
10 {
11     public class UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
   ↪  IConverter<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
   ↪  EqualityComparer<TLink>.Default;

15         private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;

17         public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
   ↪  TLink> powerOf2ToUnaryNumberConverter)
18             : 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         }

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         }

52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         private static void SetBit(ref TLink target, int powerOf2Index) => target =
   ↪  (Integer<TLink>)((Integer<TLink>)target | 1UL << powerOf2Index); // Should be
   ↪  Math.Or(target, Math.ShiftLeft(One, powerOf2Index))
54     }
55 }
```

./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>,
       ↪  IPropertiesOperator<TLink, TLink, TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
12
13         public PropertiesOperator(ILinks<TLink> links) : base(links) { }
14
15         public TLink GetValue(TLink @object, TLink property)
16         {
17             var objectProperty = Links.SearchOrDefault(@object, property);
18             if (_equalityComparer.Equals(objectProperty, default))
19             {
20                 return default;
21             }
22             var valueLink = Links.All(Links.Constants.Any, objectProperty).SingleOrDefault();
23             if (valueLink == null)
24             {
25                 return default;
26             }
27             return Links.GetTarget(valueLink[Links.Constants.IndexPart]);
28         }
29
30         public void SetValue(TLink @object, TLink property, TLink value)
31         {
32             var objectProperty = Links.GetOrCreate(@object, property);
33             Links.DeleteMany(Links.AllIndices(Links.Constants.Any, objectProperty));
34             Links.GetOrCreate(objectProperty, value);
35         }
36     }
37 }
```

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

```
1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.PropertyOperators
7  {
8      public class PropertyOperator<TLink> : LinksOperatorBase<TLink>, IPropertyOperator<TLink,
       ↪  TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
11
12         private readonly TLink _propertyMarker;
13         private readonly TLink _propertyValueMarker;
14
15         public PropertyOperator(ILinks<TLink> links, TLink propertyMarker, TLink
           ↪  propertyValueMarker) : base(links)
16         {
17             _propertyMarker = propertyMarker;
18             _propertyValueMarker = propertyValueMarker;
19         }
20
21         public TLink Get(TLink link)
22         {
23             var property = Links.SearchOrDefault(link, _propertyMarker);
24             var container = GetContainer(property);
25             var value = GetValue(container);
26             return value;
27         }
28
29         private TLink GetContainer(TLink property)
30         {
31             var valueContainer = default(TLink);
32             if (_equalityComparer.Equals(property, default))
33             {
34                 return valueContainer;
35             }
36             var constants = Links.Constants;
37             var countinueConstant = constants.Continue;
38             var breakConstant = constants.Break;
39             var anyConstant = constants.Any;
40             var query = new Link<TLink>(anyConstant, property, anyConstant);
```

```csharp
                    Links.Each(candidate =>
                    {
                        var candidateTarget = Links.GetTarget(candidate);
                        var valueTarget = Links.GetTarget(candidateTarget);
                        if (_equalityComparer.Equals(valueTarget, _propertyValueMarker))
                        {
                            valueContainer = Links.GetIndex(candidate);
                            return breakConstant;
                        }
                        return countinueConstant;
                    }, query);
                    return valueContainer;
                }

                private TLink GetValue(TLink container) => _equalityComparer.Equals(container, default)
                    ? default : Links.GetTarget(container);

                public void Set(TLink link, TLink value)
                {
                    var property = Links.GetOrCreate(link, _propertyMarker);
                    var container = GetContainer(property);
                    if (_equalityComparer.Equals(container, default))
                    {
                        Links.GetOrCreate(property, value);
                    }
                    else
                    {
                        Links.Update(container, property, value);
                    }
                }
            }
        }
    }
```

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

```csharp
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using Platform.Disposables;
using Platform.Singletons;
using Platform.Collections.Arrays;
using Platform.Numbers;
using Platform.Unsafe;
using Platform.Memory;
using Platform.Data.Exceptions;
using Platform.Data.Constants;
using static Platform.Numbers.Arithmetic;

#pragma warning disable 0649
#pragma warning disable 169
#pragma warning disable 618
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

// ReSharper disable StaticMemberInGenericType
// ReSharper disable BuiltInTypeReferenceStyle
// ReSharper disable MemberCanBePrivate.Local
// ReSharper disable UnusedMember.Local

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    public partial class ResizableDirectMemoryLinks<TLink> : DisposableBase, ILinks<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
            EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        /// <summary>Возвращает размер одной связи в байтах.</summary>
        public static readonly int LinkSizeInBytes = Structure<Link>.Size;

        public static readonly int LinkHeaderSizeInBytes = Structure<LinksHeader>.Size;

        public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;

        private struct Link
        {
            public static readonly int SourceOffset = Marshal.OffsetOf(typeof(Link),
                nameof(Source)).ToInt32();
            public static readonly int TargetOffset = Marshal.OffsetOf(typeof(Link),
                nameof(Target)).ToInt32();
            public static readonly int LeftAsSourceOffset = Marshal.OffsetOf(typeof(Link),
                nameof(LeftAsSource)).ToInt32();
```

```csharp
44          public static readonly int RightAsSourceOffset = Marshal.OffsetOf(typeof(Link),
                ↪ nameof(RightAsSource)).ToInt32();
45          public static readonly int SizeAsSourceOffset = Marshal.OffsetOf(typeof(Link),
                ↪ nameof(SizeAsSource)).ToInt32();
46          public static readonly int LeftAsTargetOffset = Marshal.OffsetOf(typeof(Link),
                ↪ nameof(LeftAsTarget)).ToInt32();
47          public static readonly int RightAsTargetOffset = Marshal.OffsetOf(typeof(Link),
                ↪ nameof(RightAsTarget)).ToInt32();
48          public static readonly int SizeAsTargetOffset = Marshal.OffsetOf(typeof(Link),
                ↪ nameof(SizeAsTarget)).ToInt32();
49
50          public TLink Source;
51          public TLink Target;
52          public TLink LeftAsSource;
53          public TLink RightAsSource;
54          public TLink SizeAsSource;
55          public TLink LeftAsTarget;
56          public TLink RightAsTarget;
57          public TLink SizeAsTarget;
58
59          [MethodImpl(MethodImplOptions.AggressiveInlining)]
60          public static TLink GetSource(IntPtr pointer) => (pointer +
                ↪ SourceOffset).GetValue<TLink>();
61          [MethodImpl(MethodImplOptions.AggressiveInlining)]
62          public static TLink GetTarget(IntPtr pointer) => (pointer +
                ↪ TargetOffset).GetValue<TLink>();
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          public static TLink GetLeftAsSource(IntPtr pointer) => (pointer +
                ↪ LeftAsSourceOffset).GetValue<TLink>();
65          [MethodImpl(MethodImplOptions.AggressiveInlining)]
66          public static TLink GetRightAsSource(IntPtr pointer) => (pointer +
                ↪ RightAsSourceOffset).GetValue<TLink>();
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();
```

```csharp
            public static readonly int FreeLinksOffset = Marshal.OffsetOf(typeof(LinksHeader),
            ↪ nameof(FreeLinks)).ToInt32();
            public static readonly int FirstFreeLinkOffset =
            ↪ Marshal.OffsetOf(typeof(LinksHeader), nameof(FirstFreeLink)).ToInt32();
            public static readonly int FirstAsSourceOffset =
            ↪ Marshal.OffsetOf(typeof(LinksHeader), nameof(FirstAsSource)).ToInt32();
            public static readonly int FirstAsTargetOffset =
            ↪ Marshal.OffsetOf(typeof(LinksHeader), nameof(FirstAsTarget)).ToInt32();
            public static readonly int LastFreeLinkOffset =
            ↪ Marshal.OffsetOf(typeof(LinksHeader), nameof(LastFreeLink)).ToInt32();

            public TLink AllocatedLinks;
            public TLink ReservedLinks;
            public TLink FreeLinks;
            public TLink FirstFreeLink;
            public TLink FirstAsSource;
            public TLink FirstAsTarget;
            public TLink LastFreeLink;
            public TLink Reserved8;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetAllocatedLinks(IntPtr pointer) => (pointer +
            ↪ AllocatedLinksOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetReservedLinks(IntPtr pointer) => (pointer +
            ↪ ReservedLinksOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetFreeLinks(IntPtr pointer) => (pointer +
            ↪ FreeLinksOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetFirstFreeLink(IntPtr pointer) => (pointer +
            ↪ FirstFreeLinkOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetFirstAsSource(IntPtr pointer) => (pointer +
            ↪ FirstAsSourceOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetFirstAsTarget(IntPtr pointer) => (pointer +
            ↪ FirstAsTargetOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetLastFreeLink(IntPtr pointer) => (pointer +
            ↪ LastFreeLinkOffset).GetValue<TLink>();

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static IntPtr GetFirstAsSourcePointer(IntPtr pointer) => pointer +
            ↪ FirstAsSourceOffset;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static IntPtr GetFirstAsTargetPointer(IntPtr pointer) => pointer +
            ↪ FirstAsTargetOffset;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetAllocatedLinks(IntPtr pointer, TLink value) => (pointer +
            ↪ AllocatedLinksOffset).SetValue(value);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetReservedLinks(IntPtr pointer, TLink value) => (pointer +
            ↪ ReservedLinksOffset).SetValue(value);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetFreeLinks(IntPtr pointer, TLink value) => (pointer +
            ↪ FreeLinksOffset).SetValue(value);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetFirstFreeLink(IntPtr pointer, TLink value) => (pointer +
            ↪ FirstFreeLinkOffset).SetValue(value);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetFirstAsSource(IntPtr pointer, TLink value) => (pointer +
            ↪ FirstAsSourceOffset).SetValue(value);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetFirstAsTarget(IntPtr pointer, TLink value) => (pointer +
            ↪ FirstAsTargetOffset).SetValue(value);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetLastFreeLink(IntPtr pointer, TLink value) => (pointer +
            ↪ LastFreeLinkOffset).SetValue(value);
        }

        private readonly long _memoryReservationStep;

        private readonly IResizableDirectMemory _memory;
        private IntPtr _header;
        private IntPtr _links;
```

```csharp
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 пусть 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
       ↪  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)
       ↪  * LinkSizeInBytes) + LinkHeaderSizeInBytes;
200            // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
201            LinksHeader.SetReservedLinks(_header, (Integer<TLink>)((_memory.ReservedCapacity -
       ↪  LinkHeaderSizeInBytes) / LinkSizeInBytes));
202        }
203
204        [MethodImpl(MethodImplOptions.AggressiveInlining)]
205        public TLink Count(IList<TLink> restrictions)
206        {
207            // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
208            if (restrictions.Count == 0)
209            {
210                return Total;
211            }
212            if (restrictions.Count == 1)
213            {
214                var index = restrictions[Constants.IndexPart];
215                if (_equalityComparer.Equals(index, Constants.Any))
216                {
217                    return Total;
218                }
219                return Exists(index) ? Integer<TLink>.One : Integer<TLink>.Zero;
220            }
221            if (restrictions.Count == 2)
222            {
223                var index = restrictions[Constants.IndexPart];
```

```csharp
                    var value = restrictions[1];
                    if (_equalityComparer.Equals(index, Constants.Any))
                    {
                        if (_equalityComparer.Equals(value, Constants.Any))
                        {
                            return Total; // Any - как отсутствие ограничения
                        }
                        return Add(_sourcesTreeMethods.CountUsages(value),
                          ↪  _targetsTreeMethods.CountUsages(value));
                    }
                    else
                    {
                        if (!Exists(index))
                        {
                            return Integer<TLink>.Zero;
                        }
                        if (_equalityComparer.Equals(value, Constants.Any))
                        {
                            return Integer<TLink>.One;
                        }
                        var storedLinkValue = GetLinkUnsafe(index);
                        if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
                            _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
                        {
                            return Integer<TLink>.One;
                        }
                        return Integer<TLink>.Zero;
                    }
                }
                if (restrictions.Count == 3)
                {
                    var index = restrictions[Constants.IndexPart];
                    var source = restrictions[Constants.SourcePart];
                    var target = restrictions[Constants.TargetPart];

                    if (_equalityComparer.Equals(index, Constants.Any))
                    {
                        if (_equalityComparer.Equals(source, Constants.Any) &&
                          ↪  _equalityComparer.Equals(target, Constants.Any))
                        {
                            return Total;
                        }
                        else if (_equalityComparer.Equals(source, Constants.Any))
                        {
                            return _targetsTreeMethods.CountUsages(target);
                        }
                        else if (_equalityComparer.Equals(target, Constants.Any))
                        {
                            return _sourcesTreeMethods.CountUsages(source);
                        }
                        else //if(source != Any && target != Any)
                        {
                            // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
                            var link = _sourcesTreeMethods.Search(source, target);
                            return _equalityComparer.Equals(link, Constants.Null) ?
                              ↪  Integer<TLink>.Zero : Integer<TLink>.One;
                        }
                    }
                    else
                    {
                        if (!Exists(index))
                        {
                            return Integer<TLink>.Zero;
                        }
                        if (_equalityComparer.Equals(source, Constants.Any) &&
                          ↪  _equalityComparer.Equals(target, Constants.Any))
                        {
                            return Integer<TLink>.One;
                        }
                        var storedLinkValue = GetLinkUnsafe(index);
                        if (!_equalityComparer.Equals(source, Constants.Any) &&
                          ↪  !_equalityComparer.Equals(target, Constants.Any))
                        {
                            if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source) &&
                                _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
                            {
                                return Integer<TLink>.One;
                            }
                        }
```

```csharp
                    return Integer<TLink>.Zero;
                }
                var value = default(TLink);
                if (_equalityComparer.Equals(source, Constants.Any))
                {
                    value = target;
                }
                if (_equalityComparer.Equals(target, Constants.Any))
                {
                    value = source;
                }
                if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
                    _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
                {
                    return Integer<TLink>.One;
                }
                return Integer<TLink>.Zero;
            }
        }
        throw new NotSupportedException("Другие размеры и способы ограничений не
        ↪ поддерживаются.");
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
    {
        if (restrictions.Count == 0)
        {
            for (TLink link = Integer<TLink>.One; _comparer.Compare(link,
            ↪ (Integer<TLink>)LinksHeader.GetAllocatedLinks(_header)) <= 0; link =
            ↪ Increment(link))
            {
                if (Exists(link) && _equalityComparer.Equals(handler(GetLinkStruct(link)),
                ↪ Constants.Break))
                {
                    return Constants.Break;
                }
            }

            return Constants.Continue;
        }
        if (restrictions.Count == 1)
        {
            var index = restrictions[Constants.IndexPart];
            if (_equalityComparer.Equals(index, Constants.Any))
            {
                return Each(handler, ArrayPool<TLink>.Empty);
            }
            if (!Exists(index))
            {
                return Constants.Continue;
            }
            return handler(GetLinkStruct(index));
        }
        if (restrictions.Count == 2)
        {
            var index = restrictions[Constants.IndexPart];
            var value = restrictions[1];
            if (_equalityComparer.Equals(index, Constants.Any))
            {
                if (_equalityComparer.Equals(value, Constants.Any))
                {
                    return Each(handler, ArrayPool<TLink>.Empty);
                }
                if (_equalityComparer.Equals(Each(handler, new[] { index, value,
                ↪ Constants.Any }), Constants.Break))
                {
                    return Constants.Break;
                }
                return Each(handler, new[] { index, Constants.Any, value });
            }
            else
            {
                if (!Exists(index))
                {
                    return Constants.Continue;
                }
                if (_equalityComparer.Equals(value, Constants.Any))
```

```
370                         {
371                             return handler(GetLinkStruct(index));
372                         }
373                         var storedLinkValue = GetLinkUnsafe(index);
374                         if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
375                             _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
376                         {
377                             return handler(GetLinkStruct(index));
378                         }
379                         return Constants.Continue;
380                     }
381                 }
382             if (restrictions.Count == 3)
383             {
384                 var index = restrictions[Constants.IndexPart];
385                 var source = restrictions[Constants.SourcePart];
386                 var target = restrictions[Constants.TargetPart];
387                 if (_equalityComparer.Equals(index, Constants.Any))
388                 {
389                     if (_equalityComparer.Equals(source, Constants.Any) &&
                        ↪ _equalityComparer.Equals(target, Constants.Any))
390                     {
391                         return Each(handler, ArrayPool<TLink>.Empty);
392                     }
393                     else if (_equalityComparer.Equals(source, Constants.Any))
394                     {
395                         return _targetsTreeMethods.EachUsage(target, handler);
396                     }
397                     else if (_equalityComparer.Equals(target, Constants.Any))
398                     {
399                         return _sourcesTreeMethods.EachUsage(source, handler);
400                     }
401                     else //if(source != Any && target != Any)
402                     {
403                         var link = _sourcesTreeMethods.Search(source, target);
404                         return _equalityComparer.Equals(link, Constants.Null) ?
                            ↪ 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) &&
421                             _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
422                         {
423                             return handler(GetLinkStruct(index));
424                         }
425                         return Constants.Continue;
426                     }
427                     var value = default(TLink);
428                     if (_equalityComparer.Equals(source, Constants.Any))
429                     {
430                         value = target;
431                     }
432                     if (_equalityComparer.Equals(target, Constants.Any))
433                     {
434                         value = source;
435                     }
436                     if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
437                         _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
438                     {
439                         return handler(GetLinkStruct(index));
440                     }
441                     return Constants.Continue;
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),
            ↪   linkIndex);
473              }
474              return linkIndex;
475          }
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,
        ↪   linkIndex);
486
487          /// <remarks>
488          /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
        ↪   пространство
489          /// </remarks>
490          public TLink Create()
491          {
492              var freeLink = LinksHeader.GetFirstFreeLink(_header);
493              if (!_equalityComparer.Equals(freeLink, Constants.Null))
494              {
495                  _unusedLinksListMethods.Detach(freeLink);
496              }
497              else
498              {
499                  if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
            ↪   Constants.MaxPossibleIndex) > 0)
500                  {
501                      throw new
                ↪   LinksLimitReachedException((Integer<TLink>)Constants.MaxPossibleIndex);
502                  }
503                  if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
            ↪   Decrement(LinksHeader.GetReservedLinks(_header))) >= 0)
504                  {
505                      _memory.ReservedCapacity += _memoryReservationStep;
506                      SetPointers(_memory);
507                      LinksHeader.SetReservedLinks(_header,
                ↪   (Integer<TLink>)(_memory.ReservedCapacity / LinkSizeInBytes));
```

```
508                }
509                LinksHeader.SetAllocatedLinks(_header,
       ↪   Increment(LinksHeader.GetAllocatedLinks(_header)));
510                _memory.UsedCapacity += LinkSizeInBytes;
511                freeLink = LinksHeader.GetAllocatedLinks(_header);
512            }
513            return freeLink;
514        }
515
516        public void Delete(TLink link)
517        {
518            if (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) < 0)
519            {
520                _unusedLinksListMethods.AttachAsFirst(link);
521            }
522            else if (_equalityComparer.Equals(link, LinksHeader.GetAllocatedLinks(_header)))
523            {
524                LinksHeader.SetAllocatedLinks(_header,
       ↪   Decrement(LinksHeader.GetAllocatedLinks(_header)));
525                _memory.UsedCapacity -= LinkSizeInBytes;
526                // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
       ↪   пока не дойдём до первой существующей связи
527                // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
528                while ((_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
       ↪   Integer<TLink>.Zero) > 0) &&
       ↪   IsUnusedLink(LinksHeader.GetAllocatedLinks(_header)))
529                {
530                    _unusedLinksListMethods.Detach(LinksHeader.GetAllocatedLinks(_header));
531                    LinksHeader.SetAllocatedLinks(_header,
       ↪   Decrement(LinksHeader.GetAllocatedLinks(_header)));
532                    _memory.UsedCapacity -= LinkSizeInBytes;
533                }
534            }
535        }
536
537        /// <remarks>
538        /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
       ↪   адрес реально поменялся
539        ///
540        /// Указатель this.links может быть в том же месте,
541        /// так как 0-я связь не используется и имеет такой же размер как Header,
542        /// поэтому header размещается в том же месте, что и 0-я связь
543        /// </remarks>
544        private void SetPointers(IDirectMemory memory)
545        {
546            if (memory == null)
547            {
548                _links = IntPtr.Zero;
549                _header = _links;
550                _unusedLinksListMethods = null;
551                _targetsTreeMethods = null;
552                _unusedLinksListMethods = null;
553            }
554            else
555            {
556                _links = memory.Pointer;
557                _header = _links;
558                _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
559                _targetsTreeMethods = new LinksTargetsTreeMethods(this);
560                _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
561            }
562        }
563
564        [MethodImpl(MethodImplOptions.AggressiveInlining)]
565        private bool Exists(TLink link)
566            => (_comparer.Compare(link, Constants.MinPossibleIndex) >= 0)
567            && (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) <= 0)
568            && !IsUnusedLink(link);
569
570        [MethodImpl(MethodImplOptions.AggressiveInlining)]
571        private bool IsUnusedLink(TLink link)
572            => _equalityComparer.Equals(LinksHeader.GetFirstFreeLink(_header), link)
573            || (_equalityComparer.Equals(Link.GetSizeAsSource(GetLinkUnsafe(link)),
       ↪   Constants.Null)
574            && !_equalityComparer.Equals(Link.GetSource(GetLinkUnsafe(link)), Constants.Null));
575
576        #region DisposableBase
577
578        protected override bool AllowMultipleDisposeCalls => true;
```

```
579
580        protected override void Dispose(bool manual, bool wasDisposed)
581        {
582            if (!wasDisposed)
583            {
584                SetPointers(null);
585                _memory.DisposeIfPossible();
586            }
587        }
588
589        #endregion
590    }
591 }
```

./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 +
                  ↪  LinksHeader.FirstFreeLinkOffset).GetValue<TLink>();
23
24             protected override TLink GetLast() => (_header +
                  ↪  LinksHeader.LastFreeLinkOffset).GetValue<TLink>();
25
26             protected override TLink GetPrevious(TLink element) =>
                  ↪  (_links.GetElement(LinkSizeInBytes, element) +
                  ↪  Link.SourceOffset).GetValue<TLink>();
27
28             protected override TLink GetNext(TLink element) =>
                  ↪  (_links.GetElement(LinkSizeInBytes, element) +
                  ↪  Link.TargetOffset).GetValue<TLink>();
29
30             protected override TLink GetSize() => (_header +
                  ↪  LinksHeader.FreeLinksOffset).GetValue<TLink>();
31
32             protected override void SetFirst(TLink element) => (_header +
                  ↪  LinksHeader.FirstFreeLinkOffset).SetValue(element);
33
34             protected override void SetLast(TLink element) => (_header +
                  ↪  LinksHeader.LastFreeLinkOffset).SetValue(element);
35
36             protected override void SetPrevious(TLink element, TLink previous) =>
                  ↪  (_links.GetElement(LinkSizeInBytes, element) +
                  ↪  Link.SourceOffset).SetValue(previous);
37
38             protected override void SetNext(TLink element, TLink next) =>
                  ↪  (_links.GetElement(LinkSizeInBytes, element) + Link.TargetOffset).SetValue(next);
39
40             protected override void SetSize(TLink size) => (_header +
                  ↪  LinksHeader.FreeLinksOffset).SetValue(size);
41         }
42     }
43 }
```

./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;
```

```csharp
using Platform.Collections.Methods.Trees;
using Platform.Data.Constants;

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

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    partial class ResizableDirectMemoryLinks<TLink>
    {
        private abstract class LinksTreeMethodsBase :
            SizedAndThreadedAVLBalancedTreeMethods<TLink>
        {
            private readonly ResizableDirectMemoryLinks<TLink> _memory;
            private readonly LinksCombinedConstants<TLink, TLink, int> _constants;
            protected readonly IntPtr Links;
            protected readonly IntPtr Header;

            protected LinksTreeMethodsBase(ResizableDirectMemoryLinks<TLink> memory)
            {
                Links = memory._links;
                Header = memory._header;
                _memory = memory;
                _constants = memory.Constants;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected abstract TLink GetTreeRoot();

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected abstract TLink GetBasePartValue(TLink link);

            public TLink this[TLink index]
            {
                get
                {
                    var root = GetTreeRoot();
                    if (GreaterOrEqualThan(index, GetSize(root)))
                    {
                        return GetZero();
                    }
                    while (!EqualToZero(root))
                    {
                        var left = GetLeftOrDefault(root);
                        var leftSize = GetSizeOrZero(left);
                        if (LessThan(index, leftSize))
                        {
                            root = left;
                            continue;
                        }
                        if (IsEquals(index, leftSize))
                        {
                            return root;
                        }
                        root = GetRightOrDefault(root);
                        index = Subtract(index, Increment(leftSize));
                    }
                    return GetZero(); // TODO: Impossible situation exception (only if tree
                        structure broken)
                }
            }

            // TODO: Return indices range instead of references count
            public TLink CountUsages(TLink link)
            {
                var root = GetTreeRoot();
                var total = GetSize(root);
                var totalRightIgnore = GetZero();
                while (!EqualToZero(root))
                {
                    var @base = GetBasePartValue(root);
                    if (LessOrEqualThan(@base, link))
                    {
                        root = GetRightOrDefault(root);
                    }
                    else
                    {
                        totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
                        root = GetLeftOrDefault(root);
                    }
                }
```

```csharp
                    root = GetTreeRoot();
                    var totalLeftIgnore = GetZero();
                    while (!EqualToZero(root))
                    {
                        var @base = GetBasePartValue(root);
                        if (GreaterOrEqualThan(@base, link))
                        {
                            root = GetLeftOrDefault(root);
                        }
                        else
                        {
                            totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));

                            root = GetRightOrDefault(root);
                        }
                    }
                    return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
                }

                public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
                {
                    var root = GetTreeRoot();
                    if (EqualToZero(root))
                    {
                        return _constants.Continue;
                    }
                    TLink first = GetZero(), current = root;
                    while (!EqualToZero(current))
                    {
                        var @base = GetBasePartValue(current);
                        if (GreaterOrEqualThan(@base, link))
                        {
                            if (IsEquals(@base, link))
                            {
                                first = current;
                            }
                            current = GetLeftOrDefault(current);
                        }
                        else
                        {
                            current = GetRightOrDefault(current);
                        }
                    }
                    if (!EqualToZero(first))
                    {
                        current = first;
                        while (true)
                        {
                            if (IsEquals(handler(_memory.GetLinkStruct(current)), _constants.Break))
                            {
                                return _constants.Break;
                            }
                            current = GetNext(current);
                            if (EqualToZero(current) || !IsEquals(GetBasePartValue(current), link))
                            {
                                break;
                            }
                        }
                    }
                    return _constants.Continue;
                }

                protected override void PrintNodeValue(TLink node, StringBuilder sb)
                {
                    sb.Append(' ');
                    sb.Append((Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SourceOffset).GetValue<TLink>());
                    sb.Append('-');
                    sb.Append('>');
                    sb.Append((Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.TargetOffset).GetValue<TLink>());
                }
            }

        private class LinksSourcesTreeMethods : LinksTreeMethodsBase
        {
            public LinksSourcesTreeMethods(ResizableDirectMemoryLinks<TLink> memory)
                : base(memory)
```

```csharp
                {
                }

            protected override IntPtr GetLeftPointer(TLink node) =>
                Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsSourceOffset;

            protected override IntPtr GetRightPointer(TLink node) =>
                Links.GetElement(LinkSizeInBytes, node) + Link.RightAsSourceOffset;

            protected override TLink GetLeftValue(TLink node) =>
                (Links.GetElement(LinkSizeInBytes, node) +
                Link.LeftAsSourceOffset).GetValue<TLink>();

            protected override TLink GetRightValue(TLink node) =>
                (Links.GetElement(LinkSizeInBytes, node) +
                Link.RightAsSourceOffset).GetValue<TLink>();

            protected override TLink GetSize(TLink node)
            {
                var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    Link.SizeAsSourceOffset).GetValue<TLink>();
                return Bit.PartialRead(previousValue, 5, -5);
            }

            protected override void SetLeft(TLink node, TLink left) =>
                (Links.GetElement(LinkSizeInBytes, node) +
                Link.LeftAsSourceOffset).SetValue(left);

            protected override void SetRight(TLink node, TLink right) =>
                (Links.GetElement(LinkSizeInBytes, node) +
                Link.RightAsSourceOffset).SetValue(right);

            protected override void SetSize(TLink node, TLink size)
            {
                var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    Link.SizeAsSourceOffset).GetValue<TLink>();
                (Links.GetElement(LinkSizeInBytes, node) +
                    Link.SizeAsSourceOffset).SetValue(Bit.PartialWrite(previousValue, size, 5,
                    -5));
            }

            protected override bool GetLeftIsChild(TLink node)
            {
                var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    Link.SizeAsSourceOffset).GetValue<TLink>();
                return (Integer<TLink>)Bit.PartialRead(previousValue, 4, 1);
            }

            protected override void SetLeftIsChild(TLink node, bool value)
            {
                var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    Link.SizeAsSourceOffset).GetValue<TLink>();
                var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 4,
                    1);
                (Links.GetElement(LinkSizeInBytes, node) +
                    Link.SizeAsSourceOffset).SetValue(modified);
            }

            protected override bool GetRightIsChild(TLink node)
            {
                var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    Link.SizeAsSourceOffset).GetValue<TLink>();
                return (Integer<TLink>)Bit.PartialRead(previousValue, 3, 1);
            }

            protected override void SetRightIsChild(TLink node, bool value)
            {
                var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    Link.SizeAsSourceOffset).GetValue<TLink>();
                var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 3,
                    1);
                (Links.GetElement(LinkSizeInBytes, node) +
                    Link.SizeAsSourceOffset).SetValue(modified);
            }

            protected override sbyte GetBalance(TLink node)
            {
```

```csharp
216                    var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
          ↪  Link.SizeAsSourceOffset).GetValue<TLink>();
217                    var value = (ulong)(Integer<TLink>)Bit.PartialRead(previousValue, 0, 3);
218                    var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
          ↪  124 : value & 3);
219                    return unpackedValue;
220                }
221
222            protected override void SetBalance(TLink node, sbyte value)
223                {
224                    var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
          ↪  Link.SizeAsSourceOffset).GetValue<TLink>();
225                    var packagedValue = (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) | value &
          ↪  3);
226                    var modified = Bit.PartialWrite(previousValue, packagedValue, 0, 3);
227                    (Links.GetElement(LinkSizeInBytes, node) +
          ↪  Link.SizeAsSourceOffset).SetValue(modified);
228                }
229
230            protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
231                {
232                    var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
          ↪  Link.SourceOffset).GetValue<TLink>();
233                    var secondSource = (Links.GetElement(LinkSizeInBytes, second) +
          ↪  Link.SourceOffset).GetValue<TLink>();
234                    return LessThan(firstSource, secondSource) ||
235                            (IsEquals(firstSource, secondSource) &&
                                LessThan((Links.GetElement(LinkSizeInBytes, first) +
          ↪  Link.TargetOffset).GetValue<TLink>(),
          ↪  (Links.GetElement(LinkSizeInBytes, second) +
          ↪  Link.TargetOffset).GetValue<TLink>()));
236                }
237
238            protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
239                {
240                    var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
          ↪  Link.SourceOffset).GetValue<TLink>();
241                    var secondSource = (Links.GetElement(LinkSizeInBytes, second) +
          ↪  Link.SourceOffset).GetValue<TLink>();
242                    return GreaterThan(firstSource, secondSource) ||
243                            (IsEquals(firstSource, secondSource) &&
                                GreaterThan((Links.GetElement(LinkSizeInBytes, first) +
          ↪  Link.TargetOffset).GetValue<TLink>(),
          ↪  (Links.GetElement(LinkSizeInBytes, second) +
          ↪  Link.TargetOffset).GetValue<TLink>()));
244                }
245
246            protected override TLink GetTreeRoot() => (Header +
          ↪  LinksHeader.FirstAsSourceOffset).GetValue<TLink>();
247
248            protected override TLink GetBasePartValue(TLink link) =>
          ↪  (Links.GetElement(LinkSizeInBytes, link) + Link.SourceOffset).GetValue<TLink>();
249
250            /// <summary>
251            /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
          ↪  (концом)
252            /// по дереву (индексу) связей, отсортированному по Source, а затем по Target.
253            /// </summary>
254            /// <param name="source">Индекс связи, которая является началом на искомой
          ↪  связи.</param>
255            /// <param name="target">Индекс связи, которая является концом на искомой
          ↪  связи.</param>
256            /// <returns>Индекс искомой связи.</returns>
257            public TLink Search(TLink source, TLink target)
258                {
259                    var root = GetTreeRoot();
260                    while (!EqualToZero(root))
261                    {
262                        var rootSource = (Links.GetElement(LinkSizeInBytes, root) +
          ↪  Link.SourceOffset).GetValue<TLink>();
263                        var rootTarget = (Links.GetElement(LinkSizeInBytes, root) +
          ↪  Link.TargetOffset).GetValue<TLink>();
264                        if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
          ↪  node.Key < root.Key
265                        {
266                            root = GetLeftOrDefault(root);
```

```
                    }
                    else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget))
                    ↪  // node.Key > root.Key
                    {
                        root = GetRightOrDefault(root);
                    }
                    else // node.Key == root.Key
                    {
                        return root;
                    }
                }
                return GetZero();
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget, TLink
            ↪  secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
            ↪  (IsEquals(firstSource, secondSource) && LessThan(firstTarget, secondTarget));

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget, TLink
            ↪  secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
            ↪  (IsEquals(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
        }

        private class LinksTargetsTreeMethods : LinksTreeMethodsBase
        {
            public LinksTargetsTreeMethods(ResizableDirectMemoryLinks<TLink> memory)
                : base(memory)
            {
            }

            protected override IntPtr GetLeftPointer(TLink node) =>
            ↪  Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsTargetOffset;

            protected override IntPtr GetRightPointer(TLink node) =>
            ↪  Links.GetElement(LinkSizeInBytes, node) + Link.RightAsTargetOffset;

            protected override TLink GetLeftValue(TLink node) =>
            ↪  (Links.GetElement(LinkSizeInBytes, node) +
            ↪  Link.LeftAsTargetOffset).GetValue<TLink>();

            protected override TLink GetRightValue(TLink node) =>
            ↪  (Links.GetElement(LinkSizeInBytes, node) +
            ↪  Link.RightAsTargetOffset).GetValue<TLink>();

            protected override TLink GetSize(TLink node)
            {
                var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                ↪  Link.SizeAsTargetOffset).GetValue<TLink>();
                return Bit.PartialRead(previousValue, 5, -5);
            }

            protected override void SetLeft(TLink node, TLink left) =>
            ↪  (Links.GetElement(LinkSizeInBytes, node) +
            ↪  Link.LeftAsTargetOffset).SetValue(left);

            protected override void SetRight(TLink node, TLink right) =>
            ↪  (Links.GetElement(LinkSizeInBytes, node) +
            ↪  Link.RightAsTargetOffset).SetValue(right);

            protected override void SetSize(TLink node, TLink size)
            {
                var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                ↪  Link.SizeAsTargetOffset).GetValue<TLink>();
                (Links.GetElement(LinkSizeInBytes, node) +
                ↪  Link.SizeAsTargetOffset).SetValue(Bit.PartialWrite(previousValue, size, 5,
                ↪  -5));
            }

            protected override bool GetLeftIsChild(TLink node)
            {
                var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                ↪  Link.SizeAsTargetOffset).GetValue<TLink>();
                return (Integer<TLink>)Bit.PartialRead(previousValue, 4, 1);
            }
```

```
324    protected override void SetLeftIsChild(TLink node, bool value)
325    {
326        var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
           ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
327        var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 4,
           ↪ 1);
328        (Links.GetElement(LinkSizeInBytes, node) +
           ↪ Link.SizeAsTargetOffset).SetValue(modified);
329    }
330
331    protected override bool GetRightIsChild(TLink node)
332    {
333        var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
           ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
334        return (Integer<TLink>)Bit.PartialRead(previousValue, 3, 1);
335    }
336
337    protected override void SetRightIsChild(TLink node, bool value)
338    {
339        var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
           ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
340        var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 3,
           ↪ 1);
341        (Links.GetElement(LinkSizeInBytes, node) +
           ↪ Link.SizeAsTargetOffset).SetValue(modified);
342    }
343
344    protected override sbyte GetBalance(TLink node)
345    {
346        var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
           ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
347        var value = (ulong)(Integer<TLink>)Bit.PartialRead(previousValue, 0, 3);
348        var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
           ↪ 124 : value & 3);
349        return unpackedValue;
350    }
351
352    protected override void SetBalance(TLink node, sbyte value)
353    {
354        var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
           ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
355        var packagedValue = (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) | value &
           ↪ 3);
356        var modified = Bit.PartialWrite(previousValue, packagedValue, 0, 3);
357        (Links.GetElement(LinkSizeInBytes, node) +
           ↪ Link.SizeAsTargetOffset).SetValue(modified);
358    }
359
360    protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
361    {
362        var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
           ↪ Link.TargetOffset).GetValue<TLink>();
363        var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
           ↪ Link.TargetOffset).GetValue<TLink>();
364        return LessThan(firstTarget, secondTarget) ||
365            (IsEquals(firstTarget, secondTarget) &&
                ↪ LessThan((Links.GetElement(LinkSizeInBytes, first) +
                ↪ Link.SourceOffset).GetValue<TLink>(),
                ↪ (Links.GetElement(LinkSizeInBytes, second) +
                ↪ Link.SourceOffset).GetValue<TLink>()));
366    }
367
368    protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
369    {
370        var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
           ↪ Link.TargetOffset).GetValue<TLink>();
371        var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
           ↪ Link.TargetOffset).GetValue<TLink>();
372        return GreaterThan(firstTarget, secondTarget) ||
373            (IsEquals(firstTarget, secondTarget) &&
                ↪ GreaterThan((Links.GetElement(LinkSizeInBytes, first) +
                ↪ Link.SourceOffset).GetValue<TLink>(),
                ↪ (Links.GetElement(LinkSizeInBytes, second) +
                ↪ Link.SourceOffset).GetValue<TLink>()));
374    }
375
```

```
376          protected override TLink GetTreeRoot() => (Header +
             ↪  LinksHeader.FirstAsTargetOffset).GetValue<TLink>();

378          protected override TLink GetBasePartValue(TLink link) =>
             ↪  (Links.GetElement(LinkSizeInBytes, link) + Link.TargetOffset).GetValue<TLink>();
379      }
380   }
381 }
```

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

```csharp
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 рекомедуется 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
68         private UnusedLinksListMethods _unusedLinksListMethods;
69
70         /// <summary>
```

```csharp
        /// Возвращает общее число связей находящихся в хранилище.
        /// </summary>
        private id Total => _header->AllocatedLinks - _header->FreeLinks;

        // TODO: Дать возможность переопределять в конструкторе
        public LinksCombinedConstants<id, id, int> Constants { get; }

        public UInt64ResizableDirectMemoryLinks(string address) : this(address,
        ↪  DefaultLinksSizeStep) { }

        /// <summary>
        /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
        ↪  минимальным шагом расширения базы данных.
        /// </summary>
        /// <param name="address">Полный пусть к файлу базы данных.</param>
        /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
        ↪  байтах.</param>
        public UInt64ResizableDirectMemoryLinks(string address, long memoryReservationStep) :
        ↪  this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
        ↪  memoryReservationStep) { }

        public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
        ↪  DefaultLinksSizeStep) { }

        public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
        ↪  memoryReservationStep)
        {
            Constants = Default<LinksCombinedConstants<id, id, int>>.Instance;
            _memory = memory;
            _memoryReservationStep = memoryReservationStep;
            if (memory.ReservedCapacity < memoryReservationStep)
            {
                memory.ReservedCapacity = memoryReservationStep;
            }
            SetPointers(_memory);
            // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
            _memory.UsedCapacity = ((long)_header->AllocatedLinks * sizeof(Link)) +
            ↪  sizeof(LinksHeader);
            // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
            _header->ReservedLinks = (id)((_memory.ReservedCapacity - sizeof(LinksHeader)) /
            ↪  sizeof(Link));
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public id Count(IList<id> restrictions)
        {
            // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
            if (restrictions.Count == 0)
            {
                return Total;
            }
            if (restrictions.Count == 1)
            {
                var index = restrictions[Constants.IndexPart];
                if (index == Constants.Any)
                {
                    return Total;
                }
                return Exists(index) ? 1UL : 0UL;
            }
            if (restrictions.Count == 2)
            {
                var index = restrictions[Constants.IndexPart];
                var value = restrictions[1];
                if (index == Constants.Any)
                {
                    if (value == Constants.Any)
                    {
                        return Total; // Any - как отсутствие ограничения
                    }
                    return _sourcesTreeMethods.CountUsages(value)
                        + _targetsTreeMethods.CountUsages(value);
                }
                else
                {
                    if (!Exists(index))
                    {
                        return 0;
```

```csharp
140                     }
141                     if (value == Constants.Any)
142                     {
143                         return 1;
144                     }
145                     var storedLinkValue = GetLinkUnsafe(index);
146                     if (storedLinkValue->Source == value ||
147                         storedLinkValue->Target == value)
148                     {
149                         return 1;
150                     }
151                     return 0;
152                 }
153             }
154             if (restrictions.Count == 3)
155             {
156                 var index = restrictions[Constants.IndexPart];
157                 var source = restrictions[Constants.SourcePart];
158                 var target = restrictions[Constants.TargetPart];
159                 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 ? 0UL : 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.");
```

```csharp
            }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public id Each(Func<IList<id>, id> handler, IList<id> restrictions)
        {
            if (restrictions.Count == 0)
            {
                for (id link = 1; link <= _header->AllocatedLinks; link++)
                {
                    if (Exists(link))
                    {
                        if (handler(GetLinkStruct(link)) == Constants.Break)
                        {
                            return Constants.Break;
                        }
                    }
                }
                return Constants.Continue;
            }
            if (restrictions.Count == 1)
            {
                var index = restrictions[Constants.IndexPart];
                if (index == Constants.Any)
                {
                    return Each(handler, ArrayPool<ulong>.Empty);
                }
                if (!Exists(index))
                {
                    return Constants.Continue;
                }
                return handler(GetLinkStruct(index));
            }
            if (restrictions.Count == 2)
            {
                var index = restrictions[Constants.IndexPart];
                var value = restrictions[1];
                if (index == Constants.Any)
                {
                    if (value == Constants.Any)
                    {
                        return Each(handler, ArrayPool<ulong>.Empty);
                    }
                    if (Each(handler, new[] { index, value, Constants.Any }) == Constants.Break)
                    {
                        return Constants.Break;
                    }
                    return Each(handler, new[] { index, Constants.Any, value });
                }
                else
                {
                    if (!Exists(index))
                    {
                        return Constants.Continue;
                    }
                    if (value == Constants.Any)
                    {
                        return handler(GetLinkStruct(index));
                    }
                    var storedLinkValue = GetLinkUnsafe(index);
                    if (storedLinkValue->Source == value ||
                        storedLinkValue->Target == value)
                    {
                        return handler(GetLinkStruct(index));
                    }
                    return Constants.Continue;
                }
            }
            if (restrictions.Count == 3)
            {
                var index = restrictions[Constants.IndexPart];
                var source = restrictions[Constants.SourcePart];
                var target = restrictions[Constants.TargetPart];
                if (index == Constants.Any)
                {
                    if (source == Constants.Any && target == Constants.Any)
                    {
                        return Each(handler, ArrayPool<ulong>.Empty);
                    }
```

```
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 :
                     ↪   handler(GetLinkStruct(link));
308              }
309          }
310          else
311          {
312              if (!Exists(index))
313              {
314                  return Constants.Continue;
315              }
316              if (source == Constants.Any && target == Constants.Any)
317              {
318                  return handler(GetLinkStruct(index));
319              }
320              var storedLinkValue = GetLinkUnsafe(index);
321              if (source != Constants.Any && target != Constants.Any)
322              {
323                  if (storedLinkValue->Source == source &&
324                      storedLinkValue->Target == target)
325                  {
326                      return handler(GetLinkStruct(index));
327                  }
328                  return Constants.Continue;
329              }
330              var value = default(id);
331              if (source == Constants.Any)
332              {
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(IList<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));
```

```csharp
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) => &_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 += _memoryReservationStep;
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)
```

```csharp
                    while (_header->AllocatedLinks > 0 && IsUnusedLink(_header->AllocatedLinks))
                    {
                        _unusedLinksListMethods.Detach(_header->AllocatedLinks);
                        _header->AllocatedLinks--;
                        _memory.UsedCapacity -= sizeof(Link);
                    }
                }
            }

            /// <remarks>
            /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
            ↪    адрес реально поменялся
            ///
            /// Указатель this.links может быть в том же месте,
            /// так как 0-я связь не используется и имеет такой же размер как Header,
            /// поэтому header размещается в том же месте, что и 0-я связь
            /// </remarks>
            private void SetPointers(IResizableDirectMemory memory)
            {
                if (memory == null)
                {
                    _header = null;
                    _links = null;
                    _unusedLinksListMethods = null;
                    _targetsTreeMethods = null;
                    _unusedLinksListMethods = null;
                }
                else
                {
                    _header = (LinksHeader*)(void*)memory.Pointer;
                    _links = (Link*)(void*)memory.Pointer;
                    _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
                    _targetsTreeMethods = new LinksTargetsTreeMethods(this);
                    _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
                }
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private bool Exists(id link) => link >= Constants.MinPossibleIndex && link <=
            ↪    _header->AllocatedLinks && !IsUnusedLink(link);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private bool IsUnusedLink(id link) => _header->FirstFreeLink == link
                                                  || (_links[link].SizeAsSource == Constants.Null &&
                                                  ↪    _links[link].Source != Constants.Null);

            #region Disposable

            protected override bool AllowMultipleDisposeCalls => true;

            protected override void Dispose(bool manual, bool wasDisposed)
            {
                if (!wasDisposed)
                {
                    SetPointers(null);
                    _memory.DisposeIfPossible();
                }
            }

            #endregion
        }
    }
```

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

```csharp
using Platform.Collections.Methods.Lists;

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

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    unsafe partial class UInt64ResizableDirectMemoryLinks
    {
        private class UnusedLinksListMethods : CircularDoublyLinkedListMethods<ulong>
        {
            private readonly Link* _links;
            private readonly LinksHeader* _header;

            public UnusedLinksListMethods(Link* links, LinksHeader* header)
            {
                _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                     {
```

```csharp
                            if (handler(_memory.GetLinkStruct(current)) == _constants.Break)
                            {
                                return _constants.Break;
                            }
                            current = GetNext(current);
                            if (current == 0 || GetBasePartValue(current) != link)
                            {
                                break;
                            }
                        }
                    }
                    return _constants.Continue;
                }

                protected override void PrintNodeValue(ulong node, StringBuilder sb)
                {
                    sb.Append(' ');
                    sb.Append(Links[node].Source);
                    sb.Append('-');
                    sb.Append('>');
                    sb.Append(Links[node].Target);
                }
            }

            private class LinksSourcesTreeMethods : LinksTreeMethodsBase
            {
                public LinksSourcesTreeMethods(UInt64ResizableDirectMemoryLinks memory)
                    : base(memory)
                {
                }

                protected override IntPtr GetLeftPointer(ulong node) => new
                ↪   IntPtr(&Links[node].LeftAsSource);

                protected override IntPtr GetRightPointer(ulong node) => new
                ↪   IntPtr(&Links[node].RightAsSource);

                protected override ulong GetLeftValue(ulong node) => Links[node].LeftAsSource;

                protected override ulong GetRightValue(ulong node) => Links[node].RightAsSource;

                protected override ulong GetSize(ulong node)
                {
                    var previousValue = Links[node].SizeAsSource;
                    //return Math.PartialRead(previousValue, 5, -5);
                    return (previousValue & 4294967264) >> 5;
                }

                protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource
                ↪   = left;

                protected override void SetRight(ulong node, ulong right) =>
                ↪   Links[node].RightAsSource = right;

                protected override void SetSize(ulong node, ulong size)
                {
                    var previousValue = Links[node].SizeAsSource;
                    //var modified = Math.PartialWrite(previousValue, size, 5, -5);
                    var modified = (previousValue & 31) | ((size & 134217727) << 5);
                    Links[node].SizeAsSource = modified;
                }

                protected override bool GetLeftIsChild(ulong node)
                {
                    var previousValue = Links[node].SizeAsSource;
                    //return (Integer)Math.PartialRead(previousValue, 4, 1);
                    return (previousValue & 16) >> 4 == 1UL;
                }

                protected override void SetLeftIsChild(ulong node, bool value)
                {
                    var previousValue = Links[node].SizeAsSource;
                    //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 4, 1);
                    var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
                    Links[node].SizeAsSource = modified;
                }

                protected override bool GetRightIsChild(ulong node)
                {
```

```csharp
                var previousValue = Links[node].SizeAsSource;
                //return (Integer)Math.PartialRead(previousValue, 3, 1);
                return (previousValue & 8) >> 3 == 1UL;
            }

        protected override void SetRightIsChild(ulong node, bool value)
            {
                var previousValue = Links[node].SizeAsSource;
                //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 3, 1);
                var modified = (previousValue & 4294967287) | ((value ? 1UL : 0UL) << 3);
                Links[node].SizeAsSource = modified;
            }

        protected override sbyte GetBalance(ulong node)
            {
                var previousValue = Links[node].SizeAsSource;
                //var value = Math.PartialRead(previousValue, 0, 3);
                var value = previousValue & 7;
                var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
                ↪  124 : value & 3);
                return unpackedValue;
            }

        protected override void SetBalance(ulong node, sbyte value)
            {
                var previousValue = Links[node].SizeAsSource;
                var packagedValue = (ulong)((((byte)value >> 5) & 4) | value & 3);
                //var modified = Math.PartialWrite(previousValue, packagedValue, 0, 3);
                var modified = (previousValue & 4294967288) | (packagedValue & 7);
                Links[node].SizeAsSource = modified;
            }

        protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
            => Links[first].Source < Links[second].Source ||
              (Links[first].Source == Links[second].Source && Links[first].Target <
              ↪  Links[second].Target);

        protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
            => Links[first].Source > Links[second].Source ||
              (Links[first].Source == Links[second].Source && Links[first].Target >
              ↪  Links[second].Target);

        protected override ulong GetTreeRoot() => Header->FirstAsSource;

        protected override ulong GetBasePartValue(ulong link) => Links[link].Source;

        /// <summary>
        /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
        ↪  (концом)
        /// по дереву (индексу) связей, отсортированному по Source, а затем по Target.
        /// </summary>
        /// <param name="source">Индекс связи, которая является началом на искомой
        ↪  связи.</param>
        /// <param name="target">Индекс связи, которая является концом на искомой
        ↪  связи.</param>
        /// <returns>Индекс искомой связи.</returns>
        public ulong Search(ulong source, ulong target)
            {
                var root = Header->FirstAsSource;
                while (root != 0)
                {
                    var rootSource = Links[root].Source;
                    var rootTarget = Links[root].Target;
                    if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                    ↪  node.Key < root.Key
                    {
                        root = GetLeftOrDefault(root);
                    }
                    else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget))
                    ↪  // node.Key > root.Key
                    {
                        root = GetRightOrDefault(root);
                    }
                    else // node.Key == root.Key
                    {
                        return root;
                    }
                }
```

```csharp
                    return 0;
                }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(ulong node)
        {
            Links[node].LeftAsSource = 0UL;
            Links[node].RightAsSource = 0UL;
            Links[node].SizeAsSource = 0UL;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetZero() => 0UL;

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool ValueEqualToZero(IntPtr pointer) =>
            ↪ *(ulong*)pointer.ToPointer() == 0UL;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool EqualToZero(ulong value) => value == 0UL;

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThanZero(ulong value) => value > 0UL;

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Add(ulong first, ulong second) => first + second;
```

```csharp
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override ulong Subtract(ulong first, ulong second) => first - second;
        }

        private class LinksTargetsTreeMethods : LinksTreeMethodsBase
        {
            public LinksTargetsTreeMethods(UInt64ResizableDirectMemoryLinks memory)
                : base(memory)
            {
            }

            //protected override IntPtr GetLeft(ulong node) => new
            //    IntPtr(&Links[node].LeftAsTarget);

            //protected override IntPtr GetRight(ulong node) => new
            //    IntPtr(&Links[node].RightAsTarget);

            //protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;

            //protected override void SetLeft(ulong node, ulong left) =>
            //    Links[node].LeftAsTarget = left;

            //protected override void SetRight(ulong node, ulong right) =>
            //    Links[node].RightAsTarget = right;

            //protected override void SetSize(ulong node, ulong size) =>
            //    Links[node].SizeAsTarget = size;

            protected override IntPtr GetLeftPointer(ulong node) => new
                IntPtr(&Links[node].LeftAsTarget);

            protected override IntPtr GetRightPointer(ulong node) => new
                IntPtr(&Links[node].RightAsTarget);

            protected override ulong GetLeftValue(ulong node) => Links[node].LeftAsTarget;

            protected override ulong GetRightValue(ulong node) => Links[node].RightAsTarget;

            protected override ulong GetSize(ulong node)
            {
                var previousValue = Links[node].SizeAsTarget;
                //return Math.PartialRead(previousValue, 5, -5);
                return (previousValue & 4294967264) >> 5;
            }

            protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget
                = left;

            protected override void SetRight(ulong node, ulong right) =>
                Links[node].RightAsTarget = right;

            protected override void SetSize(ulong node, ulong size)
            {
                var previousValue = Links[node].SizeAsTarget;
                //var modified = Math.PartialWrite(previousValue, size, 5, -5);
                var modified = (previousValue & 31) | ((size & 134217727) << 5);
                Links[node].SizeAsTarget = modified;
            }

            protected override bool GetLeftIsChild(ulong node)
            {
                var previousValue = Links[node].SizeAsTarget;
                //return (Integer)Math.PartialRead(previousValue, 4, 1);
                return (previousValue & 16) >> 4 == 1UL;
                // TODO: Check if this is possible to use
                //var nodeSize = GetSize(node);
                //var left = GetLeftValue(node);
                //var leftSize = GetSizeOrZero(left);
                //return leftSize > 0 && nodeSize > leftSize;
            }

            protected override void SetLeftIsChild(ulong node, bool value)
            {
                var previousValue = Links[node].SizeAsTarget;
                //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 4, 1);
                var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
                Links[node].SizeAsTarget = modified;
```

```csharp
                }

                protected override bool GetRightIsChild(ulong node)
                {
                    var previousValue = Links[node].SizeAsTarget;
                    //return (Integer)Math.PartialRead(previousValue, 3, 1);
                    return (previousValue & 8) >> 3 == 1UL;
                    // TODO: Check if this is possible to use
                    //var nodeSize = GetSize(node);
                    //var right = GetRightValue(node);
                    //var rightSize = GetSizeOrZero(right);
                    //return rightSize > 0 && nodeSize > rightSize;
                }

                protected override void SetRightIsChild(ulong node, bool value)
                {
                    var previousValue = Links[node].SizeAsTarget;
                    //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 3, 1);
                    var modified = (previousValue & 4294967287) | ((value ? 1UL : 0UL) << 3);
                    Links[node].SizeAsTarget = modified;
                }

                protected override sbyte GetBalance(ulong node)
                {
                    var previousValue = Links[node].SizeAsTarget;
                    //var value = Math.PartialRead(previousValue, 0, 3);
                    var value = previousValue & 7;
                    var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
                    ↪   124 : value & 3);
                    return unpackedValue;
                }

                protected override void SetBalance(ulong node, sbyte value)
                {
                    var previousValue = Links[node].SizeAsTarget;
                    var packagedValue = (ulong)((((byte)value >> 5) & 4) | value & 3);
                    //var modified = Math.PartialWrite(previousValue, packagedValue, 0, 3);
                    var modified = (previousValue & 4294967288) | (packagedValue & 7);
                    Links[node].SizeAsTarget = modified;
                }

                protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
                    => Links[first].Target < Links[second].Target ||
                       (Links[first].Target == Links[second].Target && Links[first].Source <
                       ↪   Links[second].Source);

                protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
                    => Links[first].Target > Links[second].Target ||
                       (Links[first].Target == Links[second].Target && Links[first].Source >
                       ↪   Links[second].Source);

                protected override ulong GetTreeRoot() => Header->FirstAsTarget;

                protected override ulong GetBasePartValue(ulong link) => Links[link].Target;

                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                protected override void ClearNode(ulong node)
                {
                    Links[node].LeftAsTarget = 0UL;
                    Links[node].RightAsTarget = 0UL;
                    Links[node].SizeAsTarget = 0UL;
                }
            }
        }
    }
```

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

```csharp
using System.Collections.Generic;

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

namespace Platform.Data.Doublets.Sequences.Converters
{
    public class BalancedVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
    {
        public BalancedVariantConverter(ILinks<TLink> links) : base(links) { }

        public override TLink Convert(IList<TLink> sequence)
        {
```

```
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
                    ↪  generics) but will be possible with Sigil
26                  var halvedSequence = new TLink[(length / 2) + (length % 2)];
27                  HalveSequence(halvedSequence, sequence, length);
28                  sequence = halvedSequence;
29                  length = halvedSequence.Length;
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
           ↪  Links на этапе сжатия.
17      ///     А именно будет создаваться временный список пар необходимых для выполнения сжатия, в
           ↪  таком случае тип значения элемента массива может быть любым, как char так и ulong.
18      ///     Как только список/словарь пар был выявлен можно разом выполнить создание всех этих
           ↪  пар, а так же разом выполнить замену.
19      /// </remarks>
20      public class CompressingConverter<TLink> : LinksListToSequenceConverterBase<TLink>
21      {
22          private static readonly LinksCombinedConstants<bool, TLink, long> _constants =
               ↪  Default<LinksCombinedConstants<bool, TLink, long>>.Instance;
23          private static readonly EqualityComparer<TLink> _equalityComparer =
               ↪  EqualityComparer<TLink>.Default;
24          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
25
26          private readonly IConverter<IList<TLink>, TLink> _baseConverter;
27          private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
28          private readonly TLink _minFrequencyToCompress;
29          private readonly bool _doInitialFrequenciesIncrement;
30          private Doublet<TLink> _maxDoublet;
31          private LinkFrequency<TLink> _maxDoubletData;
```

```
32
33          private struct HalfDoublet
34          {
35              public TLink Element;
36              public LinkFrequency<TLink> DoubletData;
37
38              public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
39              {
40                  Element = element;
41                  DoubletData = doubletData;
42              }
43
44              public override string ToString() => $"{Element}: ({DoubletData})";
45          }
46
47          public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
      ↪   baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
48              : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One, true)
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 stackallow 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
```

```csharp
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) &&
                                 ↪  _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
152                            {
153                                if (r > 0)
154                                {
155                                    var previous = copy[w - 1].Element;
156                                    copy[w - 1].DoubletData.DecrementFrequency();
157                                    copy[w - 1].DoubletData =
                                     ↪  _doubletFrequenciesCache.IncrementFrequency(previous,
                                     ↪  maxDoubletReplacementLink);
158                                }
159                                if (r < oldLengthMinusTwo)
160                                {
161                                    var next = copy[r + 2].Element;
162                                    copy[r + 1].DoubletData.DecrementFrequency();
163                                    copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(ma↓
                                     ↪  xDoubletReplacementLink,
                                     ↪  next);
164                                }
165                                copy[w++].Element = maxDoubletReplacementLink;
166                                r++;
167                                newLength--;
168                            }
169                            else
170                            {
171                                copy[w++] = copy[r];
172                            }
173                        }
174                        if (w < newLength)
```

```csharp
                    {
                        copy[w] = copy[r];
                    }
                    oldLength = newLength;
                    ResetMaxDoublet();
                    UpdateMaxDoublet(copy, newLength);
                }
                return newLength;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private void ResetMaxDoublet()
            {
                _maxDoublet = new Doublet<TLink>();
                _maxDoubletData = new LinkFrequency<TLink>();
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
            {
                Doublet<TLink> doublet = default;
                for (var i = 1; i < length; i++)
                {
                    doublet.Source = copy[i - 1].Element;
                    doublet.Target = copy[i].Element;
                    UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
                }
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
            {
                var frequency = data.Frequency;
                var maxFrequency = _maxDoubletData.Frequency;
                //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
                //    (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
                //    compression string data (and gives collisions quickly) */ _maxDoublet.Source +
                //    _maxDoublet.Target)))
                if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
                    (_comparer.Compare(maxFrequency, frequency) < 0 ||
                        (_equalityComparer.Equals(maxFrequency, frequency) &&
                        _comparer.Compare(Arithmetic.Add(doublet.Source, doublet.Target),
                        Arithmetic.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /* gives
                        better stability and better compression on sequent data and even on rundom
                        numbers data (but gives collisions anyway) */
                {
                    _maxDoublet = doublet;
                    _maxDoubletData = data;
                }
            }
        }
    }
}
```

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

```csharp
using System.Collections.Generic;
using Platform.Interfaces;

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

namespace Platform.Data.Doublets.Sequences.Converters
{
    public abstract class LinksListToSequenceConverterBase<TLink> : IConverter<IList<TLink>,
        TLink>
    {
        protected readonly ILinks<TLink> Links;
        public LinksListToSequenceConverterBase(ILinks<TLink> links) => Links = links;
        public abstract TLink Convert(IList<TLink> source);
    }
}
```

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

```csharp
using System.Collections.Generic;
using System.Linq;
using Platform.Interfaces;

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

namespace Platform.Data.Doublets.Sequences.Converters
{
```

```csharp
        public class OptimalVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
        {
            private static readonly EqualityComparer<TLink> _equalityComparer =
                EqualityComparer<TLink>.Default;
            private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

            private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;

            public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
                sequenceToItsLocalElementLevelsConverter) : base(links)
                => _sequenceToItsLocalElementLevelsConverter =
                    sequenceToItsLocalElementLevelsConverter;

            public override TLink Convert(IList<TLink> sequence)
            {
                var length = sequence.Count;
                if (length == 1)
                {
                    return sequence[0];
                }
                var links = Links;
                if (length == 2)
                {
                    return links.GetOrCreate(sequence[0], sequence[1]);
                }
                sequence = sequence.ToArray();
                var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
                while (length > 2)
                {
                    var levelRepeat = 1;
                    var currentLevel = levels[0];
                    var previousLevel = levels[0];
                    var skipOnce = false;
                    var w = 0;
                    for (var i = 1; i < length; i++)
                    {
                        if (_equalityComparer.Equals(currentLevel, levels[i]))
                        {
                            levelRepeat++;
                            skipOnce = false;
                            if (levelRepeat == 2)
                            {
                                sequence[w] = links.GetOrCreate(sequence[i - 1], sequence[i]);
                                var newLevel = i >= length - 1 ?
                                    GetPreviousLowerThanCurrentOrCurrent(previousLevel,
                                        currentLevel) :
                                    i < 2 ?
                                    GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
                                    GetGreatestNeigbourLowerThanCurrentOrCurrent(previousLevel,
                                        currentLevel, levels[i + 1]);
                                levels[w] = newLevel;
                                previousLevel = currentLevel;
                                w++;
                                levelRepeat = 0;
                                skipOnce = true;
                            }
                            else if (i == length - 1)
                            {
                                sequence[w] = sequence[i];
                                levels[w] = levels[i];
                                w++;
                            }
                        }
                        else
                        {
                            currentLevel = levels[i];
                            levelRepeat = 1;
                            if (skipOnce)
                            {
                                skipOnce = false;
                            }
                            else
                            {
                                sequence[w] = sequence[i - 1];
                                levels[w] = levels[i - 1];
                                previousLevel = levels[w];
                                w++;
                            }
                            if (i == length - 1)
                            {
```

```
84                    sequence[w] = sequence[i];
85                    levels[w] = levels[i];
86                    w++;
87                }
88            }
89        }
90        length = w;
91    }
92    return links.GetOrCreate(sequence[0], sequence[1]);
93 }

95 private static TLink GetGreatestNeigbourLowerThanCurrentOrCurrent(TLink previous, TLink
   ↪ current, TLink next)
96 {
97     return _comparer.Compare(previous, next) > 0
98         ? _comparer.Compare(previous, current) < 0 ? previous : current
99         : _comparer.Compare(next, current) < 0 ? next : current;
100 }

102 private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
    ↪ _comparer.Compare(next, current) < 0 ? next : current;

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,
                ↪  sequenceCandidate), _links.Constants.Null);
24      }
25  }
```

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

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

```
44            {
45                right = Links.GetOrCreate(left, right);
46                left = cursor;
47            }
48            return Links.GetOrCreate(left, right);
49        }
50    }
51 }
```

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

```
1  using System.Collections.Generic;
2  using System.Linq;
3  using Platform.Interfaces;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Sequences
8  {
9      public class DuplicateSegmentsCounter<TLink> : ICounter<int>
10     {
11         private readonly IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
           ↪ _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 ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
           ↪ IList<TLink>>>
26         {
27             private readonly IListEqualityComparer<TLink> _listComparer;
28             public ItemEquilityComparer() => _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             {
```

```csharp
                    var intermediateResult = _listComparer.Compare(left.Key, right.Key);
                    if (intermediateResult == 0)
                    {
                        intermediateResult = _listComparer.Compare(left.Value, right.Value);
                    }
                    return intermediateResult;
                }
            }

        public DuplicateSegmentsProvider(ILinks<TLink> links, ISequences<TLink> sequences)
            : base(minimumStringSegmentLength: 2)
        {
            _links = links;
            _sequences = sequences;
        }

        public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
        {
            _groups = new HashSet<KeyValuePair<IList<TLink>,
            ↪  IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
            var count = _links.Count();
            _visited = new BitString((long)(Integer<TLink>)count + 1);
            _links.Each(link =>
            {
                var linkIndex = _links.GetIndex(link);
                var linkBitIndex = (long)(Integer<TLink>)linkIndex;
                if (!_visited.Get(linkBitIndex))
                {
                    var sequenceElements = new List<TLink>();
                    _sequences.EachPart(sequenceElements.AddAndReturnTrue, linkIndex);
                    if (sequenceElements.Count > 2)
                    {
                        WalkAll(sequenceElements);
                    }
                }
                return _links.Constants.Continue;
            });
            var resultList = _groups.ToList();
            var comparer = Default<ItemComparer>.Instance;
            resultList.Sort(comparer);
#if DEBUG
            foreach (var item in resultList)
            {
                PrintDuplicates(item);
            }
#endif
            return resultList;
        }

        protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
        ↪  length) => new Segment<TLink>(elements, offset, length);

        protected override void OnDublicateFound(Segment<TLink> segment)
        {
            var duplicates = CollectDuplicatesForSegment(segment);
            if (duplicates.Count > 1)
            {
                _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
                ↪  duplicates));
            }
        }

        private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
        {
            var duplicates = new List<TLink>();
            var readAsElement = new HashSet<TLink>();
            _sequences.Each(sequence =>
            {
                duplicates.Add(sequence);
                readAsElement.Add(sequence);
                return true; // Continue
            }, segment);
            if (duplicates.Any(x => _visited.Get((Integer<TLink>)x)))
            {
                return new List<TLink>();
            }
            foreach (var duplicate in duplicates)
            {
```

```
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 formatedSequenceStructure = 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(formatedSequenceStructure);
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
        ↪ between them).
13     /// TODO: Extract interface to implement frequencies storage inside Links storage
14     /// </remarks>
15     public class LinkFrequenciesCache<TLink> : LinksOperatorBase<TLink>
16     {
17         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪ EqualityComparer<TLink>.Default;
18         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
19
20         private readonly Dictionary<Doublet<TLink>, LinkFrequency<TLink>> _doubletsCache;
21         private readonly ICounter<TLink, TLink> _frequencyCounter;
22
23         public LinkFrequenciesCache(ILinks<TLink> links, ICounter<TLink, TLink> frequencyCounter)
24             : base(links)
25         {
26             _doubletsCache = new Dictionary<Doublet<TLink>, LinkFrequency<TLink>>(4096,
                ↪ DoubletComparer<TLink>.Default);
27             _frequencyCounter = frequencyCounter;
28         }
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
        public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
        {
            var doublet = new Doublet<TLink>(source, target);
            return GetFrequency(ref doublet);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
        {
            _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
            return data;
        }

        public void IncrementFrequencies(IList<TLink> sequence)
        {
            for (var i = 1; i < sequence.Count; i++)
            {
                IncrementFrequency(sequence[i - 1], sequence[i]);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
        {
            var doublet = new Doublet<TLink>(source, target);
            return IncrementFrequency(ref doublet);
        }

        public void PrintFrequencies(IList<TLink> sequence)
        {
            for (var i = 1; i < sequence.Count; i++)
            {
                PrintFrequency(sequence[i - 1], sequence[i]);
            }
        }

        public void PrintFrequency(TLink source, TLink target)
        {
            var number = GetFrequency(source, target).Frequency;
            Console.WriteLine("({0},{1}) - {2}", source, target, number);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
        {
            if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
            {
                data.IncrementFrequency();
            }
            else
            {
                var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
                data = new LinkFrequency<TLink>(Integer<TLink>.One, link);
                if (!_equalityComparer.Equals(link, default))
                {
                    data.Frequency = Arithmetic.Add(data.Frequency,
                        _frequencyCounter.Count(link));
                }
                _doubletsCache.Add(doublet, data);
            }
            return data;
        }

        public void ValidateFrequencies()
        {
            foreach (var entry in _doubletsCache)
            {
                var value = entry.Value;
                var linkIndex = value.Link;
                if (!_equalityComparer.Equals(linkIndex, default))
                {
                    var frequency = value.Frequency;
                    var count = _frequencyCounter.Count(linkIndex);
                    // TODO: Why `frequency` always greater than `count` by 1?
                    if (((_comparer.Compare(frequency, count) > 0) &&
                        (_comparer.Compare(Arithmetic.Subtract(frequency, count),
                        Integer<TLink>.One) > 0))
```

```
105                    || ((_comparer.Compare(count, frequency) > 0) &&
                       ↪ (_comparer.Compare(Arithmetic.Subtract(count, frequency),
                       ↪ Integer<TLink>.One) > 0)))
106                    {
107                        throw new InvalidOperationException("Frequencies validation failed.");
108                    }
109                }
110                //else
111                //{
112                //    if (value.Frequency > 0)
113                //    {
114                //        var frequency = value.Frequency;
115                //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
116                //        var count = _countLinkFrequency(linkIndex);
117
118                //        if ((frequency > count && frequency - count > 1) || (count > frequency
                       ↪ && count - frequency > 1))
119                //            throw new Exception("Frequencies validation failed.");
120                //    }
121                //}
122            }
123        }
124    }
125 }
```

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

```csharp
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

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

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

```csharp
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> :
     ↪   SequenceSymbolFrequencyOneOffCounter<TLink>
8        {
9            private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
10
11           public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
     ↪   ICriterionMatcher<TLink> markedSequenceMatcher, TLink sequenceLink, TLink symbol)
12               : base(links, sequenceLink, symbol)
13               => _markedSequenceMatcher = markedSequenceMatcher;
14
15           public override TLink Count()
16           {
17               if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
18               {
19                   return default;
20               }
21               return base.Count();
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 =
     ↪   EqualityComparer<TLink>.Default;
13           private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
14
15           protected readonly ILinks<TLink> _links;
16           protected readonly TLink _sequenceLink;
17           protected readonly TLink _symbol;
18           protected TLink _total;
19
20           public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink sequenceLink,
     ↪   TLink symbol)
21           {
22               _links = links;
23               _sequenceLink = sequenceLink;
24               _symbol = symbol;
25               _total = default;
26           }
27
28           public virtual TLink Count()
29           {
30               if (_comparer.Compare(_total, default) > 0)
31               {
32                   return _total;
33               }
34               StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
     ↪   IsElement, VisitElement);
35               return _total;
36           }
37
38           private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
     ↪   _links.IsPartialPoint(x); // TODO: Use SequenceElementCreteriaMatcher instead of
     ↪   IsPartialPoint
39
40           private bool VisitElement(TLink element)
41           {
42               if (_equalityComparer.Equals(element, _symbol))
43               {
44                   _total = Arithmetic.Increment(_total);
45               }
46               return true;
47           }
48       }
49   }
```

```csharp
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,
             ↪  ICriterionMatcher<TLink> markedSequenceMatcher)
13          {
14              _links = links;
15              _markedSequenceMatcher = markedSequenceMatcher;
16          }
17
18          public TLink Count(TLink argument) => new
             ↪  TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
             ↪  _markedSequenceMatcher, argument).Count();
19      }
20  }
```

```csharp
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  }
```

```csharp
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  }
```

```csharp
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,
              ↪  link, _symbol);
   53            _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
   54        }
   55
   56        private TLink EachElementHandler(IList<TLink> doublet)
   57        {
   58            var constants = _links.Constants;
   59            var doubletIndex = doublet[constants.IndexPart];
   60            if (_visits.Add(doubletIndex))
   61            {
   62                CountCore(doubletIndex);
   63            }
   64            return constants.Continue;
   65        }
   66    }
   67 }
```

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

```
    1  using System.Collections.Generic;
    2  using Platform.Interfaces;
    3
    4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
    5
    6  namespace Platform.Data.Doublets.Sequences.HeightProviders
    7  {
    8      public class CachedSequenceHeightProvider<TLink> : LinksOperatorBase<TLink>,
           ↪  ISequenceHeightProvider<TLink>
    9      {
   10          private static readonly EqualityComparer<TLink> _equalityComparer =
              ↪  EqualityComparer<TLink>.Default;
   11
   12          private readonly TLink _heightPropertyMarker;
   13          private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
   14          private readonly IConverter<TLink> _addressToUnaryNumberConverter;
   15          private readonly IConverter<TLink> _unaryNumberToAddressConverter;
   16          private readonly IPropertiesOperator<TLink, TLink, TLink> _propertyOperator;
   17
```

```
18        public CachedSequenceHeightProvider(
19            ILinks<TLink> links,
20            ISequenceHeightProvider<TLink> baseHeightProvider,
21            IConverter<TLink> addressToUnaryNumberConverter,
22            IConverter<TLink> unaryNumberToAddressConverter,
23            TLink heightPropertyMarker,
24            IPropertiesOperator<TLink, TLink, TLink> propertyOperator)
25            : base(links)
26        {
27            _heightPropertyMarker = heightPropertyMarker;
28            _baseHeightProvider = baseHeightProvider;
29            _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
30            _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
31            _propertyOperator = propertyOperator;
32        }
33
34        public TLink Get(TLink sequence)
35        {
36            TLink height;
37            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>,
       ↪   ISequenceHeightProvider<TLink>
9      {
10         private readonly ICriterionMatcher<TLink> _elementMatcher;
11
12         public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriterionMatcher<TLink>
       ↪   elementMatcher) : base(links) => _elementMatcher = elementMatcher;
13
14         public TLink Get(TLink sequence)
15         {
16             var height = default(TLink);
17             var pairOrElement = sequence;
18             while (!_elementMatcher.IsMatched(pairOrElement))
19             {
20                 pairOrElement = Links.GetTarget(pairOrElement);
21                 height = Arithmetic.Increment(height);
22             }
23             return height;
24         }
25     }
26 }
```

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

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

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

```
1  using System.Collections.Generic;
2  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
```

```csharp
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 =
             ↪  EqualityComparer<TLink>.Default;
11
12           private readonly LinkFrequenciesCache<TLink> _cache;
13
14           public CachedFrequencyIncrementingSequenceIndex(LinkFrequenciesCache<TLink> cache) =>
             ↪  _cache = cache;
15
16           public bool Add(IList<TLink> sequence)
17           {
18               var indexed = true;
19               var i = sequence.Count;
20               while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
                 ↪  { }
21               for (; i >= 1; i--)
22               {
23                   _cache.IncrementFrequency(sequence[i - 1], sequence[i]);
24               }
25               return indexed;
26           }
27
28           private bool IsIndexedWithIncrement(TLink source, TLink target)
29           {
30               var frequency = _cache.GetFrequency(source, target);
31               if (frequency == null)
32               {
33                   return false;
34               }
35               var indexed = !_equalityComparer.Equals(frequency.Frequency, default);
36               if (indexed)
37               {
38                   _cache.IncrementFrequency(source, target);
39               }
40               return indexed;
41           }
42
43           public bool MightContain(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

```csharp
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>,
         ↪  ISequenceIndex<TLink>
9        {
10           private static readonly EqualityComparer<TLink> _equalityComparer =
             ↪  EqualityComparer<TLink>.Default;
11
12           private readonly IPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
13           private readonly IIncrementer<TLink> _frequencyIncrementer;
14
```

```
15    public FrequencyIncrementingSequenceIndex(ILinks<TLink> links, IPropertyOperator<TLink,
      ↪ TLink> frequencyPropertyOperator, IIncrementer<TLink> frequencyIncrementer)
16        : base(links)
17    {
18        _frequencyPropertyOperator = frequencyPropertyOperator;
19        _frequencyIncrementer = frequencyIncrementer;
20    }
21
22    public override bool Add(IList<TLink> sequence)
23    {
24        var indexed = true;
25        var i = sequence.Count;
26        while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
          ↪ { }
27        for (; i >= 1; i--)
28        {
29            Increment(Links.GetOrCreate(sequence[i - 1], sequence[i]));
30        }
31        return indexed;
32    }
33
34    private bool IsIndexedWithIncrement(TLink source, TLink target)
35    {
36        var link = Links.SearchOrDefault(source, target);
37        var indexed = !_equalityComparer.Equals(link, default);
38        if (indexed)
39        {
40            Increment(link);
41        }
42        return indexed;
43    }
44
45    private void Increment(TLink link)
46    {
47        var previousFrequency = _frequencyPropertyOperator.Get(link);
48        var frequency = _frequencyIncrementer.Increment(previousFrequency);
49        _frequencyPropertyOperator.Set(link, frequency);
50    }
51    }
52 }
```

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

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

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

```
1  using System.Collections.Generic;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Sequences.Indexes
6  {
7      public class SequenceIndex<TLink> : LinksOperatorBase<TLink>, ISequenceIndex<TLink>
8      {
9          private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪ EqualityComparer<TLink>.Default;
10
11         public SequenceIndex(ILinks<TLink> links) : base(links) { }
12
13         public virtual bool Add(IList<TLink> sequence)
14         {
15             var indexed = true;
16             var i = sequence.Count;
```

```
17          while (--i >= 1 && (indexed =
     ↪   !_equalityComparer.Equals(Links.SearchOrDefault(sequence[i - 1], sequence[i]),
     ↪   default))) { }
18          for (; i >= 1; i--)
19          {
20              Links.GetOrCreate(sequence[i - 1], sequence[i]);
21          }
22          return indexed;
23      }
24
25      public virtual bool MightContain(IList<TLink> sequence)
26      {
27          var indexed = true;
28          var i = sequence.Count;
29          while (--i >= 1 && (indexed =
     ↪   !_equalityComparer.Equals(Links.SearchOrDefault(sequence[i - 1], sequence[i]),
     ↪   default))) { }
30          return indexed;
31      }
32  }
33 }
```

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

```
1  using System.Collections.Generic;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Sequences.Indexes
6  {
7      public class SynchronizedSequenceIndex<TLink> : ISequenceIndex<TLink>
8      {
9          private static readonly EqualityComparer<TLink> _equalityComparer =
     ↪   EqualityComparer<TLink>.Default;
10
11          private readonly ISynchronizedLinks<TLink> _links;
12
13          public SynchronizedSequenceIndex(ISynchronizedLinks<TLink> links) => _links = links;
14
15          public bool Add(IList<TLink> sequence)
16          {
17              var indexed = true;
18              var i = sequence.Count;
19              var links = _links.Unsync;
20              _links.SyncRoot.ExecuteReadOperation(() =>
21              {
22                  while (--i >= 1 && (indexed =
     ↪   !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
     ↪   sequence[i]), default))) { }
23              });
24              if (!indexed)
25              {
26                  _links.SyncRoot.ExecuteWriteOperation(() =>
27                  {
28                      for (; i >= 1; i--)
29                      {
30                          links.GetOrCreate(sequence[i - 1], sequence[i]);
31                      }
32                  });
33              }
34              return indexed;
35          }
36
37          public bool MightContain(IList<TLink> sequence)
38          {
39              var links = _links.Unsync;
40              return _links.SyncRoot.ExecuteReadOperation(() =>
41              {
42                  var indexed = true;
43                  var i = sequence.Count;
44                  while (--i >= 1 && (indexed =
     ↪   !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
     ↪   sequence[i]), default))) { }
45                  return indexed;
46              });
47          }
48      }
49 }
```

```csharp
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      /// x*y - 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      ///
48      /// Можно убрать зависимость от конкретной реализации Links,
49      /// на зависимость от абстрактного элемента, который может быть представлен несколькими
50      ↪   способами.
51      ///
52      /// Можно ли как-то сделать один общий интерфейс
53      ///
54      ///
55      /// Блокчейн и/или гит для распределённой записи транзакций.
56      ///
57      /// </remarks>
58      public partial class Sequences : ISequences<ulong> // IList<string>, IList<ulong[]> (после
      ↪   завершения реализации Sequences)
59      {
60          private static readonly LinksCombinedConstants<bool, ulong, long> _constants =
            ↪   Default<LinksCombinedConstants<bool, ulong, long>>.Instance;
61
62          /// <summary>Возвращает значение ulong, обозначающее любое количество связей.</summary>
63          public const ulong ZeroOrMany = ulong.MaxValue;
64
65          public SequencesOptions<ulong> Options;
66          public readonly SynchronizedLinks<ulong> Links;
67          public readonly ISynchronization Sync;
68
69          public Sequences(SynchronizedLinks<ulong> links)
70              : this(links, new SequencesOptions<ulong>())
71          {
72          }
73
74          public Sequences(SynchronizedLinks<ulong> links, SequencesOptions<ulong> options)
75          {
76              Links = links;
77              Sync = links.SyncRoot;
```

```csharp
                Options = options;

                Options.ValidateOptions();
                Options.InitOptions(Links);
            }

            public bool IsSequence(ulong sequence)
            {
                return Sync.ExecuteReadOperation(() =>
                {
                    if (Options.UseSequenceMarker)
                    {
                        return Options.MarkedSequenceMatcher.IsMatched(sequence);
                    }
                    return !Links.Unsync.IsPartialPoint(sequence);
                });
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private ulong GetSequenceByElements(ulong sequence)
            {
                if (Options.UseSequenceMarker)
                {
                    return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
                }
                return sequence;
            }

            private ulong GetSequenceElements(ulong sequence)
            {
                if (Options.UseSequenceMarker)
                {
                    var linkContents = new UInt64Link(Links.GetLink(sequence));
                    if (linkContents.Source == Options.SequenceMarkerLink)
                    {
                        return linkContents.Target;
                    }
                    if (linkContents.Target == Options.SequenceMarkerLink)
                    {
                        return linkContents.Source;
                    }
                }
                return sequence;
            }

            #region Count

            public ulong Count(params ulong[] sequence)
            {
                if (sequence.Length == 0)
                {
                    return Links.Count(_constants.Any, Options.SequenceMarkerLink, _constants.Any);
                }
                if (sequence.Length == 1) // Первая связь это адрес
                {
                    if (sequence[0] == _constants.Null)
                    {
                        return 0;
                    }
                    if (sequence[0] == _constants.Any)
                    {
                        return Count();
                    }
                    if (Options.UseSequenceMarker)
                    {
                        return Links.Count(_constants.Any, Options.SequenceMarkerLink, sequence[0]);
                    }
                    return Links.Exists(sequence[0]) ? 1UL : 0;
                }
                throw new NotImplementedException();
            }

            private ulong CountUsages(params ulong[] restrictions)
            {
                if (restrictions.Length == 0)
                {
                    return 0;
                }
                if (restrictions.Length == 1) // Первая связь это адрес
```

```csharp
            {
                if (restrictions[0] == _constants.Null)
                {
                    return 0;
                }
                if (Options.UseSequenceMarker)
                {
                    var elementsLink = GetSequenceElements(restrictions[0]);
                    var sequenceLink = GetSequenceByElements(elementsLink);
                    if (sequenceLink != _constants.Null)
                    {
                        return Links.Count(sequenceLink) + Links.Count(elementsLink) - 1;
                    }
                    return Links.Count(elementsLink);
                }
                return Links.Count(restrictions[0]);
            }
            throw new NotImplementedException();
        }

        #endregion

        #region Create

        public ulong Create(params ulong[] sequence)
        {
            return Sync.ExecuteWriteOperation(() =>
            {
                if (sequence.IsNullOrEmpty())
                {
                    return _constants.Null;
                }
                Links.EnsureEachLinkExists(sequence);
                return CreateCore(sequence);
            });
        }

        private ulong CreateCore(params ulong[] sequence)
        {
            if (Options.UseIndex)
            {
                Options.Index.Add(sequence);
            }
            var sequenceRoot = default(ulong);
            if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
            {
                var matches = Each(sequence);
                if (matches.Count > 0)
                {
                    sequenceRoot = matches[0];
                }
            }
            else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
            {
                return CompactCore(sequence);
            }
            if (sequenceRoot == default)
            {
                sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
            }
            if (Options.UseSequenceMarker)
            {
                Links.Unsync.CreateAndUpdate(Options.SequenceMarkerLink, sequenceRoot);
            }
            return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
        }

        #endregion

        #region Each

        public List<ulong> Each(params ulong[] sequence)
        {
            var results = new List<ulong>();
            Each(results.AddAndReturnTrue, sequence);
            return results;
        }

        public bool Each(Func<ulong, bool> handler, IList<ulong> sequence)
```

```csharp
            {
                return Sync.ExecuteReadOperation(() =>
                {
                    if (sequence.IsNullOrEmpty())
                    {
                        return true;
                    }
                    Links.EnsureEachLinkIsAnyOrExists(sequence);
                    if (sequence.Count == 1)
                    {
                        var link = sequence[0];
                        if (link == _constants.Any)
                        {
                            return Links.Unsync.Each(_constants.Any, _constants.Any, handler);
                        }
                        return handler(link);
                    }
                    if (sequence.Count == 2)
                    {
                        return Links.Unsync.Each(sequence[0], sequence[1], handler);
                    }
                    if (Options.UseIndex && !Options.Index.MightContain(sequence))
                    {
                        return false;
                    }
                    return EachCore(handler, sequence);
                });
            }

            private bool EachCore(Func<ulong, bool> handler, IList<ulong> sequence)
            {
                var matcher = new Matcher(this, sequence, new HashSet<LinkIndex>(), handler);
                // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
                //       Id.
                Func<ulong, bool> innerHandler = Options.UseSequenceMarker ? (Func<ulong,
                //       bool>)matcher.HandleFullMatchedSequence : matcher.HandleFullMatched;
                //if (sequence.Length >= 2)
                if (!StepRight(innerHandler, sequence[0], sequence[1]))
                {
                    return false;
                }
                var last = sequence.Count - 2;
                for (var i = 1; i < last; i++)
                {
                    if (!PartialStepRight(innerHandler, sequence[i], sequence[i + 1]))
                    {
                        return false;
                    }
                }
                if (sequence.Count >= 3)
                {
                    if (!StepLeft(innerHandler, sequence[sequence.Count - 2],
                    //       sequence[sequence.Count - 1]))
                    {
                        return false;
                    }
                }
                return true;
            }

            private bool PartialStepRight(Func<ulong, bool> handler, ulong left, ulong right)
            {
                return Links.Unsync.Each(_constants.Any, left, doublet =>
                {
                    if (!StepRight(handler, doublet, right))
                    {
                        return false;
                    }
                    if (left != doublet)
                    {
                        return PartialStepRight(handler, doublet, right);
                    }
                    return true;
                });
            }
```

```csharp
private bool StepRight(Func<ulong, bool> handler, ulong left, ulong right) =>
    Links.Unsync.Each(left, _constants.Any, rightStep => TryStepRightUp(handler, right,
    rightStep));

private bool TryStepRightUp(Func<ulong, bool> handler, ulong right, ulong stepFrom)
{
    var upStep = stepFrom;
    var firstSource = Links.Unsync.GetTarget(upStep);
    while (firstSource != right && firstSource != upStep)
    {
        upStep = firstSource;
        firstSource = Links.Unsync.GetSource(upStep);
    }
    if (firstSource == right)
    {
        return handler(stepFrom);
    }
    return true;
}

private bool StepLeft(Func<ulong, bool> handler, ulong left, ulong right) =>
    Links.Unsync.Each(_constants.Any, right, leftStep => TryStepLeftUp(handler, left,
    leftStep));

private bool TryStepLeftUp(Func<ulong, bool> handler, ulong left, ulong stepFrom)
{
    var upStep = stepFrom;
    var firstTarget = Links.Unsync.GetSource(upStep);
    while (firstTarget != left && firstTarget != upStep)
    {
        upStep = firstTarget;
        firstTarget = Links.Unsync.GetTarget(upStep);
    }
    if (firstTarget == left)
    {
        return handler(stepFrom);
    }
    return true;
}

#endregion

#region Update

public ulong Update(ulong[] sequence, ulong[] newSequence)
{
    if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
    {
        return _constants.Null;
    }
    if (sequence.IsNullOrEmpty())
    {
        return Create(newSequence);
    }
    if (newSequence.IsNullOrEmpty())
    {
        Delete(sequence);
        return _constants.Null;
    }
    return Sync.ExecuteWriteOperation(() =>
    {
        Links.EnsureEachLinkIsAnyOrExists(sequence);
        Links.EnsureEachLinkExists(newSequence);
        return UpdateCore(sequence, newSequence);
    });
}

private ulong UpdateCore(ulong[] sequence, ulong[] newSequence)
{
    ulong bestVariant;
    if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
        !sequence.EqualTo(newSequence))
    {
        bestVariant = CompactCore(newSequence);
    }
    else
    {
        bestVariant = CreateCore(newSequence);
    }
```

```csharp
            // TODO: Check all options only ones before loop execution
            // Возможно нужно две версии Each, возвращающий фактические последовательности и с
            ↪  маркером,
            // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
            ↪  можно получить имея только фактические последовательности.
            foreach (var variant in Each(sequence))
            {
                if (variant != bestVariant)
                {
                    UpdateOneCore(variant, bestVariant);
                }
            }
            return bestVariant;
        }

        private void UpdateOneCore(ulong sequence, ulong newSequence)
        {
            if (Options.UseGarbageCollection)
            {
                var sequenceElements = GetSequenceElements(sequence);
                var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
                var sequenceLink = GetSequenceByElements(sequenceElements);
                var newSequenceElements = GetSequenceElements(newSequence);
                var newSequenceLink = GetSequenceByElements(newSequenceElements);
                if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
                {
                    if (sequenceLink != _constants.Null)
                    {
                        Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
                    }
                    Links.Unsync.MergeUsages(sequenceElements, newSequenceElements);
                }
                ClearGarbage(sequenceElementsContents.Source);
                ClearGarbage(sequenceElementsContents.Target);
            }
            else
            {
                if (Options.UseSequenceMarker)
                {
                    var sequenceElements = GetSequenceElements(sequence);
                    var sequenceLink = GetSequenceByElements(sequenceElements);
                    var newSequenceElements = GetSequenceElements(newSequence);
                    var newSequenceLink = GetSequenceByElements(newSequenceElements);
                    if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
                    {
                        if (sequenceLink != _constants.Null)
                        {
                            Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
                        }
                        Links.Unsync.MergeUsages(sequenceElements, newSequenceElements);
                    }
                }
                else
                {
                    if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
                    {
                        Links.Unsync.MergeUsages(sequence, newSequence);
                    }
                }
            }
        }

        #endregion

        #region Delete

        public void Delete(params ulong[] sequence)
        {
            Sync.ExecuteWriteOperation(() =>
            {
                // TODO: Check all options only ones before loop execution
                foreach (var linkToDelete in Each(sequence))
                {
                    DeleteOneCore(linkToDelete);
                }
            });
        }
```

```csharp
        private void DeleteOneCore(ulong link)
        {
            if (Options.UseGarbageCollection)
            {
                var sequenceElements = GetSequenceElements(link);
                var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
                var sequenceLink = GetSequenceByElements(sequenceElements);
                if (Options.UseCascadeDelete || CountUsages(link) == 0)
                {
                    if (sequenceLink != _constants.Null)
                    {
                        Links.Unsync.Delete(sequenceLink);
                    }
                    Links.Unsync.Delete(link);
                }
                ClearGarbage(sequenceElementsContents.Source);
                ClearGarbage(sequenceElementsContents.Target);
            }
            else
            {
                if (Options.UseSequenceMarker)
                {
                    var sequenceElements = GetSequenceElements(link);
                    var sequenceLink = GetSequenceByElements(sequenceElements);
                    if (Options.UseCascadeDelete || CountUsages(link) == 0)
                    {
                        if (sequenceLink != _constants.Null)
                        {
                            Links.Unsync.Delete(sequenceLink);
                        }
                        Links.Unsync.Delete(link);
                    }
                }
                else
                {
                    if (Options.UseCascadeDelete || CountUsages(link) == 0)
                    {
                        Links.Unsync.Delete(link);
                    }
                }
            }
        }

        #endregion

        #region Compactification

        /// <remarks>
        /// bestVariant можно выбирать по максимальному числу использований,
        /// но балансированный позволяет гарантировать уникальность (если есть возможность,
        /// гарантировать его использование в других местах).
        ///
        /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
        /// </remarks>
        public ulong Compact(params ulong[] sequence)
        {
            return Sync.ExecuteWriteOperation(() =>
            {
                if (sequence.IsNullOrEmpty())
                {
                    return _constants.Null;
                }
                Links.EnsureEachLinkExists(sequence);
                return CompactCore(sequence);
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private ulong CompactCore(params ulong[] sequence) => UpdateCore(sequence, sequence);

        #endregion

        #region Garbage Collection

        /// <remarks>
        /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
        ///    определить извне или в унаследованном классе
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
        private bool IsGarbage(ulong link) => link != Options.SequenceMarkerLink &&
        ↪  !Links.Unsync.IsPartialPoint(link) && Links.Count(link) == 0;

        private void ClearGarbage(ulong link)
        {
            if (IsGarbage(link))
            {
                var contents = new UInt64Link(Links.GetLink(link));
                Links.Unsync.Delete(link);
                ClearGarbage(contents.Source);
                ClearGarbage(contents.Target);
            }
        }

        #endregion

        #region Walkers

        public bool EachPart(Func<ulong, bool> handler, ulong sequence)
        {
            return Sync.ExecuteReadOperation(() =>
            {
                var links = Links.Unsync;
                foreach (var part in Options.Walker.Walk(sequence))
                {
                    if (!handler(part))
                    {
                        return false;
                    }
                }
                return true;
            });
        }

        public class Matcher : RightSequenceWalker<ulong>
        {
            private readonly Sequences _sequences;
            private readonly IList<LinkIndex> _patternSequence;
            private readonly HashSet<LinkIndex> _linksInSequence;
            private readonly HashSet<LinkIndex> _results;
            private readonly Func<ulong, bool> _stopableHandler;
            private readonly HashSet<ulong> _readAsElements;
            private int _filterPosition;

            public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
            ↪  HashSet<LinkIndex> results, Func<LinkIndex, bool> stopableHandler,
            ↪  HashSet<LinkIndex> readAsElements = null)
                : base(sequences.Links.Unsync, new DefaultStack<ulong>())
            {
                _sequences = sequences;
                _patternSequence = patternSequence;
                _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                ↪  _constants.Any && x != ZeroOrMany));
                _results = results;
                _stopableHandler = stopableHandler;
                _readAsElements = readAsElements;
            }

            protected override bool IsElement(ulong link) => base.IsElement(link) ||
            ↪  (_readAsElements != null && _readAsElements.Contains(link)) ||
            ↪  _linksInSequence.Contains(link);

            public bool FullMatch(LinkIndex sequenceToMatch)
            {
                _filterPosition = 0;
                foreach (var part in Walk(sequenceToMatch))
                {
                    if (!FullMatchCore(part))
                    {
                        break;
                    }
                }
                return _filterPosition == _patternSequence.Count;
            }

            private bool FullMatchCore(LinkIndex element)
            {
                if (_filterPosition == _patternSequence.Count)
                {
                    _filterPosition = -2; // Длиннее чем нужно
```

```csharp
                        return false;
                    }
                    if (_patternSequence[_filterPosition] != _constants.Any
                     && element != _patternSequence[_filterPosition])
                    {
                        _filterPosition = -1;
                        return false; // Начинается/Продолжается иначе
                    }
                    _filterPosition++;
                    return true;
                }

                public void AddFullMatchedToResults(ulong sequenceToMatch)
                {
                    if (FullMatch(sequenceToMatch))
                    {
                        _results.Add(sequenceToMatch);
                    }
                }

                public bool HandleFullMatched(ulong sequenceToMatch)
                {
                    if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
                    {
                        return _stopableHandler(sequenceToMatch);
                    }
                    return true;
                }

                public bool HandleFullMatchedSequence(ulong sequenceToMatch)
                {
                    var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
                    if (sequence != _constants.Null && FullMatch(sequenceToMatch) &&
                     _results.Add(sequenceToMatch))
                    {
                        return _stopableHandler(sequence);
                    }
                    return true;
                }

                /// <remarks>
                /// TODO: Add support for LinksConstants.Any
                /// </remarks>
                public bool PartialMatch(LinkIndex sequenceToMatch)
                {
                    _filterPosition = -1;
                    foreach (var part in Walk(sequenceToMatch))
                    {
                        if (!PartialMatchCore(part))
                        {
                            break;
                        }
                    }
                    return _filterPosition == _patternSequence.Count - 1;
                }

                private bool PartialMatchCore(LinkIndex element)
                {
                    if (_filterPosition == (_patternSequence.Count - 1))
                    {
                        return false; // Нашлось
                    }
                    if (_filterPosition >= 0)
                    {
                        if (element == _patternSequence[_filterPosition + 1])
                        {
                            _filterPosition++;
                        }
                        else
                        {
                            _filterPosition = -1;
                        }
                    }
                    if (_filterPosition < 0)
                    {
                        if (element == _patternSequence[0])
                        {
                            _filterPosition = 0;
                        }
                    }
```

```
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             {
```

```csharp
                   if (sequence.IsNullOrEmpty())
                   {
                       return new ulong[0];
                   }
                   Links.EnsureEachLinkExists(sequence);
                   if (sequence.Length == 1)
                   {
                       return sequence;
                   }
                   return CreateAllVariants2Core(sequence, 0, sequence.Length - 1);
               });
        }

        private ulong[] CreateAllVariants2Core(ulong[] sequence, long startAt, long stopAt)
        {
#if DEBUG
            if ((stopAt - startAt) < 0)
            {
                throw new ArgumentOutOfRangeException(nameof(startAt), "startAt должен быть
                ↪  меньше или равен stopAt");
            }
#endif
            if ((stopAt - startAt) == 0)
            {
                return new[] { sequence[startAt] };
            }
            if ((stopAt - startAt) == 1)
            {
                return new[] { Links.Unsync.CreateAndUpdate(sequence[startAt], sequence[stopAt])
                ↪  };
            }
            var variants = new ulong[(ulong)Platform.Numbers.Math.Catalan(stopAt - startAt)];
            var last = 0;
            for (var splitter = startAt; splitter < stopAt; splitter++)
            {
                var left = CreateAllVariants2Core(sequence, startAt, splitter);
                var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
                for (var i = 0; i < left.Length; i++)
                {
                    for (var j = 0; j < right.Length; j++)
                    {
                        var variant = Links.Unsync.CreateAndUpdate(left[i], right[j]);
                        if (variant == _constants.Null)
                        {
                            throw new NotImplementedException("Creation cancellation is not
                            ↪  implemented.");
                        }
                        variants[last++] = variant;
                    }
                }
            }
            return variants;
        }

        public List<ulong> CreateAllVariants1(params ulong[] sequence)
        {
            return Sync.ExecuteWriteOperation(() =>
            {
                if (sequence.IsNullOrEmpty())
                {
                    return new List<ulong>();
                }
                Links.Unsync.EnsureEachLinkExists(sequence);
                if (sequence.Length == 1)
                {
                    return new List<ulong> { sequence[0] };
                }
                var results = new
                ↪  List<ulong>((int)Platform.Numbers.Math.Catalan(sequence.Length));
                return CreateAllVariants1Core(sequence, results);
            });
        }

        private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
        {
            if (sequence.Length == 2)
            {
                var link = Links.Unsync.CreateAndUpdate(sequence[0], sequence[1]);
```

```
104              if (link == _constants.Null)
105              {
106                  throw new NotImplementedException("Creation cancellation is not
        ↪  implemented.");
107              }
108              results.Add(link);
109              return results;
110          }
111          var innerSequenceLength = sequence.Length - 1;
112          var innerSequence = new ulong[innerSequenceLength];
113          for (var li = 0; li < innerSequenceLength; li++)
114          {
115              var link = Links.Unsync.CreateAndUpdate(sequence[li], sequence[li + 1]);
116              if (link == _constants.Null)
117              {
118                  throw new NotImplementedException("Creation cancellation is not
        ↪  implemented.");
119              }
120              for (var isi = 0; isi < li; isi++)
121              {
122                  innerSequence[isi] = sequence[isi];
123              }
124              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                          {
```

```csharp
                            innerSequence[isi] = sequence[isi + 1];
                        }
                    }
                    innerSequence[linkIndex] = doublet;
                    Each1(handler, innerSequence);
                    return _constants.Continue;
                });
            }
        }
    }

    public HashSet<ulong> EachPart(params ulong[] sequence)
    {
        var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
        EachPartCore(link =>
        {
            if (!visitedLinks.Contains(link))
            {
                visitedLinks.Add(link); // изучить почему случаются повторы
            }
            return true;
        }, sequence);
        return visitedLinks;
    }

    public void EachPart(Func<ulong, bool> handler, params ulong[] sequence)
    {
        var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
        EachPartCore(link =>
        {
            if (!visitedLinks.Contains(link))
            {
                visitedLinks.Add(link); // изучить почему случаются повторы
                return handler(link);
            }
            return true;
        }, sequence);
    }

    private void EachPartCore(Func<ulong, bool> handler, params ulong[] sequence)
    {
        if (sequence.IsNullOrEmpty())
        {
            return;
        }
        Links.EnsureEachLinkIsAnyOrExists(sequence);
        if (sequence.Length == 1)
        {
            var link = sequence[0];
            if (link > 0)
            {
                handler(link);
            }
            else
            {
                Links.Each(_constants.Any, _constants.Any, handler);
            }
        }
        else if (sequence.Length == 2)
        {
            //_links.Each(sequence[0], sequence[1], handler);
            //  o_|        x_o ...
            //  x_|         |___|
            Links.Each(sequence[1], _constants.Any, doublet =>
            {
                var match = Links.SearchOrDefault(sequence[0], doublet);
                if (match != _constants.Null)
                {
                    handler(match);
                }
                return true;
            });
            //  |_x       ... x_o
            //  |_o         |___|
            Links.Each(_constants.Any, sequence[0], doublet =>
            {
                var match = Links.SearchOrDefault(doublet, sequence[1]);
                if (match != 0)
```

```csharp
                    {
                        handler(match);
                    }
                    return true;
                });
                //          ._x o_.
                //           |___|
                PartialStepRight(x => handler(x), sequence[0], sequence[1]);
            }
            else
            {
                // TODO: Implement other variants
                return;
            }
        }

        private void PartialStepRight(Action<ulong> handler, ulong left, ulong right)
        {
            Links.Unsync.Each(_constants.Any, left, doublet =>
            {
                StepRight(handler, doublet, right);
                if (left != doublet)
                {
                    PartialStepRight(handler, doublet, right);
                }
                return true;
            });
        }

        private void StepRight(Action<ulong> handler, ulong left, ulong right)
        {
            Links.Unsync.Each(left, _constants.Any, rightStep =>
            {
                TryStepRightUp(handler, right, rightStep);
                return true;
            });
        }

        private void TryStepRightUp(Action<ulong> handler, ulong right, ulong stepFrom)
        {
            var upStep = stepFrom;
            var firstSource = Links.Unsync.GetTarget(upStep);
            while (firstSource != right && firstSource != upStep)
            {
                upStep = firstSource;
                firstSource = Links.Unsync.GetSource(upStep);
            }
            if (firstSource == right)
            {
                handler(stepFrom);
            }
        }

        // TODO: Test
        private void PartialStepLeft(Action<ulong> handler, ulong left, ulong right)
        {
            Links.Unsync.Each(right, _constants.Any, doublet =>
            {
                StepLeft(handler, left, doublet);
                if (right != doublet)
                {
                    PartialStepLeft(handler, left, doublet);
                }
                return true;
            });
        }

        private void StepLeft(Action<ulong> handler, ulong left, ulong right)
        {
            Links.Unsync.Each(_constants.Any, right, leftStep =>
            {
                TryStepLeftUp(handler, left, leftStep);
                return true;
            });
        }

        private void TryStepLeftUp(Action<ulong> handler, ulong left, ulong stepFrom)
        {
            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,
                        ↪  Links.Unsync.GetTarget,
401                          x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
                        ↪  x =>
402                          {
403                              if (filterPosition == sequence.Length)
404                              {
405                                  filterPosition = -2; // Длиннее чем нужно
406                                  return false;
407                              }
408                              if (x != sequence[filterPosition])
409                              {
410                                  filterPosition = -1;
411                                  return false; // Начинается иначе
412                              }
413                              filterPosition++;
```

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

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

public const int MaxSequenceFormatSize = 200;

public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
↪   => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
```

```csharp
486     public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
    ↪   elementToString, bool insertComma, params LinkIndex[] knownElements) =>
    ↪   Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
    ↪   elementToString, insertComma, knownElements));
487
488     private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
    ↪   Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
    ↪   LinkIndex[] knownElements)
489     {
490         var linksInSequence = new HashSet<ulong>(knownElements);
491         //var entered = new HashSet<ulong>();
492         var sb = new StringBuilder();
493         sb.Append('{');
494         if (links.Exists(sequenceLink))
495         {
496             StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
497                 x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
                    ↪   entered.AddAndReturnVoid, x => { }, entered.DoNotContains
498                 {
499                     if (insertComma && sb.Length > 1)
500                     {
501                         sb.Append(',');
502                     }
503                     //if (entered.Contains(element))
504                     //{
505                     //    sb.Append('{');
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),
                    ↪   entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
537                 {
538                     if (insertComma && sb.Length > 1)
539                     {
540                         sb.Append(',');
541                     }
542                     if (entered.Contains(element))
543                     {
544                         sb.Append('{');
545                         elementToString(sb, element);
546                         sb.Append('}');
547                     }
548                     else
```

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

```csharp
                        var results = new HashSet<ulong>();
                        for (var i = 0; i < sequence.Length; i++)
                        {
                            AllUsagesCore(sequence[i], results);
                        }
                        var filteredResults = new HashSet<ulong>();
                        var matcher = new Matcher(this, sequence, filteredResults, null);
                        matcher.AddAllPartialMatchedToResults(results);
                        return filteredResults;
                    }
                    return new HashSet<ulong>();
                });
        }

        public bool GetAllPartiallyMatchingSequences2(Func<ulong, bool> handler, params ulong[]
        ↪  sequence)
        {
            return Sync.ExecuteReadOperation(() =>
            {
                if (sequence.Length > 0)
                {
                    Links.EnsureEachLinkExists(sequence);

                    var results = new HashSet<ulong>();
                    var filteredResults = new HashSet<ulong>();
                    var matcher = new Matcher(this, sequence, filteredResults, handler);
                    for (var i = 0; i < sequence.Length; i++)
                    {
                        if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
                        {
                            return false;
                        }
                    }
                    return true;
                }
                return true;
            });
        }

        //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
        //{
        //    return Sync.ExecuteReadOperation(() =>
        //    {
        //        if (sequence.Length > 0)
        //        {
        //            _links.EnsureEachLinkIsAnyOrExists(sequence);

        //            var firstResults = new HashSet<ulong>();
        //            var lastResults = new HashSet<ulong>();

        //            var first = sequence.First(x => x != LinksConstants.Any);
        //            var last = sequence.Last(x => x != LinksConstants.Any);

        //            AllUsagesCore(first, firstResults);
        //            AllUsagesCore(last, lastResults);

        //            firstResults.IntersectWith(lastResults);

        //            //for (var i = 0; i < sequence.Length; i++)
        //            //    AllUsagesCore(sequence[i], results);

        //            var filteredResults = new HashSet<ulong>();
        //            var matcher = new Matcher(this, sequence, filteredResults, null);
        //            matcher.AddAllPartialMatchedToResults(firstResults);
        //            return filteredResults;
        //        }

        //        return new HashSet<ulong>();
        //    });
        //}

        public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
        {
            return Sync.ExecuteReadOperation(() =>
            {
                if (sequence.Length > 0)
                {
                    Links.EnsureEachLinkIsAnyOrExists(sequence);
                    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,
        ↪ IList<ulong> sequence)
722          {
723              return Sync.ExecuteReadOperation(() =>
724              {
725                  if (sequence.Count > 0)
726                  {
727                      Links.EnsureEachLinkExists(sequence);
728                      var results = new HashSet<LinkIndex>();
729                      //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,
                        ↪ readAsElements);
757                      matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
                        ↪ x)); // OrderBy is a Hack
758                      return filteredResults;
759                  }
760                  return new HashSet<ulong>();
761              });
762          }
763
764          // Does not work
765          public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
        ↪ params ulong[] sequence)
766          {
767              var visited = new HashSet<ulong>();
768              var results = new HashSet<ulong>();
769              var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
                ↪ true; }, readAsElements);
770              var last = sequence.Length - 1;
771              for (var i = 0; i < last; i++)
772              {
773                  PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
774              }
775              return results;
776          }
```

```csharp
        public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
        {
            return Sync.ExecuteReadOperation(() =>
            {
                if (sequence.Length > 0)
                {
                    Links.EnsureEachLinkExists(sequence);
                    //var firstElement = sequence[0];
                    //if (sequence.Length == 1)
                    //{
                    //    //results.Add(firstElement);
                    //    return results;
                    //}
                    //if (sequence.Length == 2)
                    //{
                    //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
                    //    //if (doublet != Doublets.Links.Null)
                    //    //    results.Add(doublet);
                    //    return results;
                    //}
                    //var lastElement = sequence[sequence.Length - 1];
                    //Func<ulong, bool> handler = x =>
                    //{
                    //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
                    //        results.Add(x);
                    //    return true;
                    //};
                    //if (sequence.Length >= 2)
                    //    StepRight(handler, sequence[0], sequence[1]);
                    //var last = sequence.Length - 2;
                    //for (var i = 1; i < last; i++)
                    //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
                    //if (sequence.Length >= 3)
                    //    StepLeft(handler, sequence[sequence.Length - 2],
                    //    sequence[sequence.Length - 1]);
                    //////if (sequence.Length == 1)
                    //////{
                    //////    throw new NotImplementedException(); // all sequences, containing
                    //////    this element?
                    //////}
                    //////if (sequence.Length == 2)
                    //////{
                    //////    var results = new List<ulong>();
                    //////    PartialStepRight(results.Add, sequence[0], sequence[1]);
                    //////    return results;
                    //////}
                    //////var matches = new List<List<ulong>>();
                    //////var last = sequence.Length - 1;
                    //////for (var i = 0; i < last; i++)
                    //////{
                    //////    var results = new List<ulong>();
                    //////    //StepRight(results.Add, sequence[i], sequence[i + 1]);
                    //////    PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
                    //////    if (results.Count > 0)
                    //////        matches.Add(results);
                    //////    else
                    //////        return results;
                    //////    if (matches.Count == 2)
                    //////    {
                    //////        var merged = new List<ulong>();
                    //////        for (var j = 0; j < matches[0].Count; j++)
                    //////            for (var k = 0; k < matches[1].Count; k++)
                    //////                CloseInnerConnections(merged.Add, matches[0][j],
                    //////                matches[1][k]);
                    //////        if (merged.Count > 0)
                    //////            matches = new List<List<ulong>> { merged };
                    //////        else
                    //////            return new List<ulong>();
                    //////    }
                    //////}
                    //////if (matches.Count > 0)
                    //////{
                    //////    var usages = new HashSet<ulong>();
                    //////    for (int i = 0; i < sequence.Length; i++)
                    //////    {
                    //////        AllUsagesCore(sequence[i], usages);
```

```csharp
                        //////      }
                        //////      //for (int i = 0; i < matches[0].Count; i++)
                        //////      //   AllUsagesCore(matches[0][i], usages);
                        //////      //usages.UnionWith(matches[0]);
                        //////      return usages.ToList();
                        //////}
                    var firstLinkUsages = new HashSet<ulong>();
                    AllUsagesCore(sequence[0], firstLinkUsages);
                    firstLinkUsages.Add(sequence[0]);
                    //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
                    ↪   sequence[0] }; // or all sequences, containing this element?
                    //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
                    ↪   1).ToList();
                    var results = new HashSet<ulong>();
                    foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
                    ↪   firstLinkUsages, 1))
                    {
                        AllUsagesCore(match, results);
                    }
                    return results.ToList();
                }
                return new List<ulong>();
            });
        }

        /// <remarks>
        /// TODO: Может потребоваться ограничение на уровень глубины рекурсии
        /// </remarks>
        public HashSet<ulong> AllUsages(ulong link)
        {
            return Sync.ExecuteReadOperation(() =>
            {
                var usages = new HashSet<ulong>();
                AllUsagesCore(link, usages);
                return usages;
            });
        }

        // При сборе всех использований (последовательностей) можно сохранять  обратный  путь  к
        ↪   той связи с которой начинался поиск (STTTSSSTT),
        // причём достаточно одного бита для хранения перехода влево или вправо
        private void AllUsagesCore(ulong link, HashSet<ulong> usages)
        {
            bool handler(ulong doublet)
            {
                if (usages.Add(doublet))
                {
                    AllUsagesCore(doublet, usages);
                }
                return true;
            }
            Links.Unsync.Each(link, _constants.Any, handler);
            Links.Unsync.Each(_constants.Any, link, handler);
        }

        public HashSet<ulong> AllBottomUsages(ulong link)
        {
            return Sync.ExecuteReadOperation(() =>
            {
                var visits = new HashSet<ulong>();
                var usages = new HashSet<ulong>();
                AllBottomUsagesCore(link, visits, usages);
                return usages;
            });
        }

        private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
        ↪   usages)
        {
            bool handler(ulong doublet)
            {
                if (visits.Add(doublet))
                {
                    AllBottomUsagesCore(doublet, visits, usages);
                }
                return true;
            }
            if (Links.Unsync.Count(_constants.Any, link) == 0)
```

```csharp
                    {
                        usages.Add(link);
                    }
                    else
                    {
                        Links.Unsync.Each(link, _constants.Any, handler);
                        Links.Unsync.Each(_constants.Any, link, handler);
                    }
                }

        public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
        {
            if (Options.UseSequenceMarker)
            {
                var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
                ↪  Options.MarkedSequenceMatcher, symbol);
                return counter.Count();
            }
            else
            {
                var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
                ↪  symbol);
                return counter.Count();
            }
        }

        private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<ulong, bool>
        ↪  outerHandler)
        {
            bool handler(ulong doublet)
            {
                if (usages.Add(doublet))
                {
                    if (!outerHandler(doublet))
                    {
                        return false;
                    }
                    if (!AllUsagesCore1(doublet, usages, outerHandler))
                    {
                        return false;
                    }
                }
                return true;
            }
            return Links.Unsync.Each(link, _constants.Any, handler)
                && Links.Unsync.Each(_constants.Any, link, handler);
        }

        public void CalculateAllUsages(ulong[] totals)
        {
            var calculator = new AllUsagesCalculator(Links, totals);
            calculator.Calculate();
        }

        public void CalculateAllUsages2(ulong[] totals)
        {
            var calculator = new AllUsagesCalculator2(Links, totals);
            calculator.Calculate();
        }

        private class AllUsagesCalculator
        {
            private readonly SynchronizedLinks<ulong> _links;
            private readonly ulong[] _totals;

            public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
            {
                _links = links;
                _totals = totals;
            }

            public void Calculate() => _links.Each(_constants.Any, _constants.Any,
            ↪  CalculateCore);

            private bool CalculateCore(ulong link)
            {
                if (_totals[link] == 0)
                {
                    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 зацикливания
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);
```

```csharp
                            // Обработка элемента
                            if (isElement(target))
                            {
                                visitLeaf(target);
                            }
                            if (isElement(source))
                            {
                                visitLeaf(source);
                            }
                            element = source;
                        }
                        else
                        {
                            stack.Push(element);
                            visitNode(element);
                            element = getTarget(element);
                        }
                    }
                }
                _totals[link]++;
                return true;
            }
        }

        private class AllUsagesCollector
        {
            private readonly ILinks<ulong> _links;
            private readonly HashSet<ulong> _usages;

            public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
            {
                _links = links;
                _usages = usages;
            }

            public bool Collect(ulong link)
            {
                if (_usages.Add(link))
                {
                    _links.Each(link, _constants.Any, Collect);
                    _links.Each(_constants.Any, link, Collect);
                }
                return true;
            }
        }

        private class AllUsagesCollector1
        {
            private readonly ILinks<ulong> _links;
            private readonly HashSet<ulong> _usages;
            private readonly ulong _continue;

            public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
            {
                _links = links;
                _usages = usages;
                _continue = _links.Constants.Continue;
            }

            public ulong Collect(IList<ulong> link)
            {
                var linkIndex = _links.GetIndex(link);
                if (_usages.Add(linkIndex))
                {
                    _links.Each(Collect, _constants.Any, linkIndex);
                }
                return _continue;
            }
        }

        private class AllUsagesCollector2
        {
            private readonly ILinks<ulong> _links;
            private readonly BitString _usages;

            public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
            {
                _links = links;
                _usages = usages;
            }
```

```csharp
            public bool Collect(ulong link)
            {
                if (_usages.Add((long)link))
                {
                    _links.Each(link, _constants.Any, Collect);
                    _links.Each(_constants.Any, link, Collect);
                }
                return true;
            }
        }

        private class AllUsagesIntersectingCollector
        {
            private readonly SynchronizedLinks<ulong> _links;
            private readonly HashSet<ulong> _intersectWith;
            private readonly HashSet<ulong> _usages;
            private readonly HashSet<ulong> _enter;

            public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
            ↪   intersectWith, HashSet<ulong> usages)
            {
                _links = links;
                _intersectWith = intersectWith;
                _usages = usages;
                _enter = new HashSet<ulong>(); // защита от зацикливания
            }

            public bool Collect(ulong link)
            {
                if (_enter.Add(link))
                {
                    if (_intersectWith.Contains(link))
                    {
                        _usages.Add(link);
                    }
                    _links.Unsync.Each(link, _constants.Any, Collect);
                    _links.Unsync.Each(_constants.Any, link, Collect);
                }
                return true;
            }
        }

        private void CloseInnerConnections(Action<ulong> handler, ulong left, ulong right)
        {
            TryStepLeftUp(handler, left, right);
            TryStepRightUp(handler, right, left);
        }

        private void AllCloseConnections(Action<ulong> handler, ulong left, ulong right)
        {
            // Direct
            if (left == right)
            {
                handler(left);
            }
            var doublet = Links.Unsync.SearchOrDefault(left, right);
            if (doublet != _constants.Null)
            {
                handler(doublet);
            }
            // Inner
            CloseInnerConnections(handler, left, right);
            // Outer
            StepLeft(handler, left, right);
            StepRight(handler, left, right);
            PartialStepRight(handler, left, right);
            PartialStepLeft(handler, left, right);
        }

        private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
        ↪   HashSet<ulong> previousMatchings, long startAt)
        {
            if (startAt >= sequence.Length) // ?
            {
                return previousMatchings;
            }
            var secondLinkUsages = new HashSet<ulong>();
            AllUsagesCore(sequence[startAt], secondLinkUsages);
```

```csharp
                        secondLinkUsages.Add(sequence[startAt]);
                        var matchings = new HashSet<ulong>();
                        //for (var i = 0; i < previousMatchings.Count; i++)
                        foreach (var secondLinkUsage in secondLinkUsages)
                        {
                            foreach (var previousMatching in previousMatchings)
                            {
                                //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
                                ↪  secondLinkUsage);
                                StepRight(matchings.AddAndReturnVoid, previousMatching, secondLinkUsage);
                                TryStepRightUp(matchings.AddAndReturnVoid, secondLinkUsage,
                                ↪  previousMatching);
                                //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
                                ↪  sequence[startAt]); // почему-то эта ошибочная запись приводит к
                                ↪  желаемым результам.
                                PartialStepRight(matchings.AddAndReturnVoid, previousMatching,
                                ↪  secondLinkUsage);
                            }
                        }
                        if (matchings.Count == 0)
                        {
                            return matchings;
                        }
                        return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
                    }

        private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
        ↪  links, params ulong[] sequence)
        {
            if (sequence == null)
            {
                return;
            }
            for (var i = 0; i < sequence.Length; i++)
            {
                if (sequence[i] != _constants.Any && sequence[i] != ZeroOrMany &&
                ↪  !links.Exists(sequence[i]))
                {
                    throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                    ↪  $"patternSequence[{i}]");
                }
            }
        }

        // Pattern Matching -> Key To Triggers
        public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
        {
            return Sync.ExecuteReadOperation(() =>
            {
                patternSequence = Simplify(patternSequence);
                if (patternSequence.Length > 0)
                {
                    EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
                    var uniqueSequenceElements = new HashSet<ulong>();
                    for (var i = 0; i < patternSequence.Length; i++)
                    {
                        if (patternSequence[i] != _constants.Any && patternSequence[i] !=
                        ↪  ZeroOrMany)
                        {
                            uniqueSequenceElements.Add(patternSequence[i]);
                        }
                    }
                    var results = new HashSet<ulong>();
                    foreach (var uniqueSequenceElement in uniqueSequenceElements)
                    {
                        AllUsagesCore(uniqueSequenceElement, results);
                    }
                    var filteredResults = new HashSet<ulong>();
                    var matcher = new PatternMatcher(this, patternSequence, filteredResults);
                    matcher.AddAllPatternMatchedToResults(results);
                    return filteredResults;
                }
                return new HashSet<ulong>();
            });
        }

        // Найти все возможные связи между указанным списком связей.
```

```csharp
            // Находит связи между всеми указанными связями в любом порядке.
            // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
            ↪  несколько раз в последовательности)
            public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
            {
                return Sync.ExecuteReadOperation(() =>
                {
                    var results = new HashSet<ulong>();
                    if (linksToConnect.Length > 0)
                    {
                        Links.EnsureEachLinkExists(linksToConnect);
                        AllUsagesCore(linksToConnect[0], results);
                        for (var i = 1; i < linksToConnect.Length; i++)
                        {
                            var next = new HashSet<ulong>();
                            AllUsagesCore(linksToConnect[i], next);
                            results.IntersectWith(next);
                        }
                    }
                    return results;
                });
            }

            public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
            {
                return Sync.ExecuteReadOperation(() =>
                {
                    var results = new HashSet<ulong>();
                    if (linksToConnect.Length > 0)
                    {
                        Links.EnsureEachLinkExists(linksToConnect);
                        var collector1 = new AllUsagesCollector(Links.Unsync, results);
                        collector1.Collect(linksToConnect[0]);
                        var next = new HashSet<ulong>();
                        for (var i = 1; i < linksToConnect.Length; i++)
                        {
                            var collector = new AllUsagesCollector(Links.Unsync, next);
                            collector.Collect(linksToConnect[i]);
                            results.IntersectWith(next);
                            next.Clear();
                        }
                    }
                    return results;
                });
            }

            public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
            {
                return Sync.ExecuteReadOperation(() =>
                {
                    var results = new HashSet<ulong>();
                    if (linksToConnect.Length > 0)
                    {
                        Links.EnsureEachLinkExists(linksToConnect);
                        var collector1 = new AllUsagesCollector(Links, results);
                        collector1.Collect(linksToConnect[0]);
                        //AllUsagesCore(linksToConnect[0], results);
                        for (var i = 1; i < linksToConnect.Length; i++)
                        {
                            var next = new HashSet<ulong>();
                            var collector = new AllUsagesIntersectingCollector(Links, results, next);
                            collector.Collect(linksToConnect[i]);
                            //AllUsagesCore(linksToConnect[i], next);
                            //results.IntersectWith(next);
                            results = next;
                        }
                    }
                    return results;
                });
            }

            public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
            {
                return Sync.ExecuteReadOperation(() =>
                {
                    var results = new BitString((long)Links.Unsync.Count() + 1); // new
                    ↪  BitArray((int)_links.Total + 1);
                    if (linksToConnect.Length > 0)
```

```csharp
                {
                    Links.EnsureEachLinkExists(linksToConnect);
                    var collector1 = new AllUsagesCollector2(Links.Unsync, results);
                    collector1.Collect(linksToConnect[0]);
                    for (var i = 1; i < linksToConnect.Length; i++)
                    {
                        var next = new BitString((long)Links.Unsync.Count() + 1); //new
                        ↪ BitArray((int)_links.Total + 1);
                        var collector = new AllUsagesCollector2(Links.Unsync, next);
                        collector.Collect(linksToConnect[i]);
                        results = results.And(next);
                    }
                }
                return results.GetSetUInt64Indices();
            });
        }

        private static ulong[] Simplify(ulong[] sequence)
        {
            // Считаем новый размер последовательности
            long newLength = 0;
            var zeroOrManyStepped = false;
            for (var i = 0; i < sequence.Length; i++)
            {
                if (sequence[i] == ZeroOrMany)
                {
                    if (zeroOrManyStepped)
                    {
                        continue;
                    }
                    zeroOrManyStepped = true;
                }
                else
                {
                    //if (zeroOrManyStepped) Is it efficient?
                    zeroOrManyStepped = false;
                }
                newLength++;
            }
            // Строим новую последовательность
            zeroOrManyStepped = false;
            var newSequence = new ulong[newLength];
            long j = 0;
            for (var i = 0; i < sequence.Length; i++)
            {
                //var current = zeroOrManyStepped;
                //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
                //if (current && zeroOrManyStepped)
                //    continue;
                //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
                //if (zeroOrManyStepped && newZeroOrManyStepped)
                //    continue;
                //zeroOrManyStepped = newZeroOrManyStepped;
                if (sequence[i] == ZeroOrMany)
                {
                    if (zeroOrManyStepped)
                    {
                        continue;
                    }
                    zeroOrManyStepped = true;
                }
                else
                {
                    //if (zeroOrManyStepped) Is it efficient?
                    zeroOrManyStepped = false;
                }
                newSequence[j++] = sequence[i];
            }
            return newSequence;
        }

        public static void TestSimplify()
        {
            var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany,
            ↪ ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
            var simplifiedSequence = Simplify(sequence);
        }

        public List<ulong> GetSimilarSequences() => new List<ulong>();
```

```csharp
public void Prediction()
{
    //_links
    //sequences
}

#region From Triplets

//public static void DeleteSequence(Link sequence)
//{
//}

public List<ulong> CollectMatchingSequences(ulong[] links)
{
    if (links.Length == 1)
    {
        throw new Exception("Подпоследовательности с одним элементом не
            поддерживаются.");
    }
    var leftBound = 0;
    var rightBound = links.Length - 1;
    var left = links[leftBound++];
    var right = links[rightBound--];
    var results = new List<ulong>();
    CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
    return results;
}

private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
    middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
{
    var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
    var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
    if (leftLinkTotalReferers <= rightLinkTotalReferers)
    {
        var nextLeftLink = middleLinks[leftBound];
        var elements = GetRightElements(leftLink, nextLeftLink);
        if (leftBound <= rightBound)
        {
            for (var i = elements.Length - 1; i >= 0; i--)
            {
                var element = elements[i];
                if (element != 0)
                {
                    CollectMatchingSequences(element, leftBound + 1, middleLinks,
                        rightLink, rightBound, ref results);
                }
            }
        }
        else
        {
            for (var i = elements.Length - 1; i >= 0; i--)
            {
                var element = elements[i];
                if (element != 0)
                {
                    results.Add(element);
                }
            }
        }
    }
    else
    {
        var nextRightLink = middleLinks[rightBound];
        var elements = GetLeftElements(rightLink, nextRightLink);
        if (leftBound <= rightBound)
        {
            for (var i = elements.Length - 1; i >= 0; i--)
            {
                var element = elements[i];
                if (element != 0)
                {
                    CollectMatchingSequences(leftLink, leftBound, middleLinks,
                        elements[i], rightBound - 1, ref results);
                }
            }
        }
    }
```

```csharp
                else
                {
                    for (var i = elements.Length - 1; i >= 0; i--)
                    {
                        var element = elements[i];
                        if (element != 0)
                        {
                            results.Add(element);
                        }
                    }
                }
            }
        }

        public ulong[] GetRightElements(ulong startLink, ulong rightLink)
        {
            var result = new ulong[5];
            TryStepRight(startLink, rightLink, result, 0);
            Links.Each(_constants.Any, startLink, couple =>
            {
                if (couple != startLink)
                {
                    if (TryStepRight(couple, rightLink, result, 2))
                    {
                        return false;
                    }
                }
                return true;
            });
            if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
            {
                result[4] = startLink;
            }
            return result;
        }

        public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
        {
            var added = 0;
            Links.Each(startLink, _constants.Any, couple =>
            {
                if (couple != startLink)
                {
                    var coupleTarget = Links.GetTarget(couple);
                    if (coupleTarget == rightLink)
                    {
                        result[offset] = couple;
                        if (++added == 2)
                        {
                            return false;
                        }
                    }
                    else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker
                        ↪    == Net.And &&
                    {
                        result[offset + 1] = couple;
                        if (++added == 2)
                        {
                            return false;
                        }
                    }
                }
                return true;
            });
            return added > 0;
        }

        public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
        {
            var result = new ulong[5];
            TryStepLeft(startLink, leftLink, result, 0);
            Links.Each(startLink, _constants.Any, couple =>
            {
                if (couple != startLink)
                {
                    if (TryStepLeft(couple, leftLink, result, 2))
                    {
                        return false;
                    }
                }
```

```
                    }
                    return true;
                });
                if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
                {
                    result[4] = leftLink;
                }
                return result;
            }

            public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
            {
                var added = 0;
                Links.Each(_constants.Any, startLink, couple =>
                {
                    if (couple != startLink)
                    {
                        var coupleSource = Links.GetSource(couple);
                        if (coupleSource == leftLink)
                        {
                            result[offset] = couple;
                            if (++added == 2)
                            {
                                return false;
                            }
                        }
                        else if (Links.GetTarget(coupleSource) == leftLink) // coupleSource.Linker
                        ↪  == Net.And &&
                        {
                            result[offset + 1] = couple;
                            if (++added == 2)
                            {
                                return false;
                            }
                        }
                    }
                    return true;
                });
                return added > 0;
            }

            #endregion

            #region Walkers

            public class PatternMatcher : RightSequenceWalker<ulong>
            {
                private readonly Sequences _sequences;
                private readonly ulong[] _patternSequence;
                private readonly HashSet<LinkIndex> _linksInSequence;
                private readonly HashSet<LinkIndex> _results;

                #region Pattern Match

                enum PatternBlockType
                {
                    Undefined,
                    Gap,
                    Elements
                }

                struct PatternBlock
                {
                    public PatternBlockType Type;
                    public long Start;
                    public long Stop;
                }

                private readonly List<PatternBlock> _pattern;
                private int _patternPosition;
                private long _sequencePosition;

                #endregion

                public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
                ↪  HashSet<LinkIndex> results)
                    : base(sequences.Links.Unsync, new DefaultStack<ulong>())
                {
                    _sequences = sequences;
                    _patternSequence = patternSequence;
```

```csharp
                    _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                    ↪  _constants.Any && x != ZeroOrMany));
                    _results = results;
                    _pattern = CreateDetailedPattern();
                }

                protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
                ↪  base.IsElement(link);

                public bool PatternMatch(LinkIndex sequenceToMatch)
                {
                    _patternPosition = 0;
                    _sequencePosition = 0;
                    foreach (var part in Walk(sequenceToMatch))
                    {
                        if (!PatternMatchCore(part))
                        {
                            break;
                        }
                    }
                    return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
                    ↪  - 1 && _pattern[_patternPosition].Start == 0);
                }

                private List<PatternBlock> CreateDetailedPattern()
                {
                    var pattern = new List<PatternBlock>();
                    var patternBlock = new PatternBlock();
                    for (var i = 0; i < _patternSequence.Length; i++)
                    {
                        if (patternBlock.Type == PatternBlockType.Undefined)
                        {
                            if (_patternSequence[i] == _constants.Any)
                            {
                                patternBlock.Type = PatternBlockType.Gap;
                                patternBlock.Start = 1;
                                patternBlock.Stop = 1;
                            }
                            else if (_patternSequence[i] == ZeroOrMany)
                            {
                                patternBlock.Type = PatternBlockType.Gap;
                                patternBlock.Start = 0;
                                patternBlock.Stop = long.MaxValue;
                            }
                            else
                            {
                                patternBlock.Type = PatternBlockType.Elements;
                                patternBlock.Start = i;
                                patternBlock.Stop = i;
                            }
                        }
                        else if (patternBlock.Type == PatternBlockType.Elements)
                        {
                            if (_patternSequence[i] == _constants.Any)
                            {
                                pattern.Add(patternBlock);
                                patternBlock = new PatternBlock
                                {
                                    Type = PatternBlockType.Gap,
                                    Start = 1,
                                    Stop = 1
                                };
                            }
                            else if (_patternSequence[i] == ZeroOrMany)
                            {
                                pattern.Add(patternBlock);
                                patternBlock = new PatternBlock
                                {
                                    Type = PatternBlockType.Gap,
                                    Start = 0,
                                    Stop = long.MaxValue
                                };
                            }
                            else
                            {
                                patternBlock.Stop = i;
                            }
                        }
                        else // patternBlock.Type == PatternBlockType.Gap
                        {
```

```
                    if (_patternSequence[i] == _constants.Any)
                    {
                        patternBlock.Start++;
                        if (patternBlock.Stop < patternBlock.Start)
                        {
                            patternBlock.Stop = patternBlock.Start;
                        }
                    }
                    else if (_patternSequence[i] == ZeroOrMany)
                    {
                        patternBlock.Stop = long.MaxValue;
                    }
                    else
                    {
                        pattern.Add(patternBlock);
                        patternBlock = new PatternBlock
                        {
                            Type = PatternBlockType.Elements,
                            Start = i,
                            Stop = i
                        };
                    }
                }
            }
            if (patternBlock.Type != PatternBlockType.Undefined)
            {
                pattern.Add(patternBlock);
            }
            return pattern;
        }

        // match: search for regexp anywhere in text
        //int match(char* regexp, char* text)
        //{
        //    do
        //    {
        //    } while (*text++ != '\0');
        //    return 0;
        //}

        // matchhere: search for regexp at beginning of text
        //int matchhere(char* regexp, char* text)
        //{
        //    if (regexp[0] == '\0')
        //        return 1;
        //    if (regexp[1] == '*')
        //        return matchstar(regexp[0], regexp + 2, text);
        //    if (regexp[0] == '$' && regexp[1] == '\0')
        //        return *text == '\0';
        //    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
        //        return matchhere(regexp + 1, text + 1);
        //    return 0;
        //}

        // matchstar: search for c*regexp at beginning of text
        //int matchstar(int c, char* regexp, char* text)
        //{
        //    do
        //    {    /* a * matches zero or more instances */
        //        if (matchhere(regexp, text))
        //            return 1;
        //    } while (*text != '\0' && (*text++ == c || c == '.'));
        //    return 0;
        //}

        //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out
        // ↪  long maximumGap)
        //{
        //    mininumGap = 0;
        //    maximumGap = 0;
        //    element = 0;
        //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
        //    {
        //        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
        //            mininumGap++;
        //        else if (_patternSequence[_patternPosition] == ZeroOrMany)
        //            maximumGap = long.MaxValue;
        //        else
        //            break;
```

```csharp
            //    }

            //    if (maximumGap < mininumGap)
            //        maximumGap = mininumGap;
            //}

            private bool PatternMatchCore(LinkIndex element)
            {
                if (_patternPosition >= _pattern.Count)
                {
                    _patternPosition = -2;
                    return false;
                }
                var currentPatternBlock = _pattern[_patternPosition];
                if (currentPatternBlock.Type == PatternBlockType.Gap)
                {
                    //var currentMatchingBlockLength = (_sequencePosition -
                    ↪  _lastMatchedBlockPosition);
                    if (_sequencePosition < currentPatternBlock.Start)
                    {
                        _sequencePosition++;
                        return true; // Двигаемся дальше
                    }
                    // Это последний блок
                    if (_pattern.Count == _patternPosition + 1)
                    {
                        _patternPosition++;
                        _sequencePosition = 0;
                        return false; // Полное соответствие
                    }
                    else
                    {
                        if (_sequencePosition > currentPatternBlock.Stop)
                        {
                            return false; // Соответствие невозможно
                        }
                        var nextPatternBlock = _pattern[_patternPosition + 1];
                        if (_patternSequence[nextPatternBlock.Start] == element)
                        {
                            if (nextPatternBlock.Start < nextPatternBlock.Stop)
                            {
                                _patternPosition++;
                                _sequencePosition = 1;
                            }
                            else
                            {
                                _patternPosition += 2;
                                _sequencePosition = 0;
                            }
                        }
                    }
                }
                else // currentPatternBlock.Type == PatternBlockType.Elements
                {
                    var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
                    if (_patternSequence[patternElementPosition] != element)
                    {
                        return false; // Соответствие невозможно
                    }
                    if (patternElementPosition == currentPatternBlock.Stop)
                    {
                        _patternPosition++;
                        _sequencePosition = 0;
                    }
                    else
                    {
                        _sequencePosition++;
                    }
                }
                return true;
                //if (_patternSequence[_patternPosition] != element)
                //    return false;
                //else
                //{
                //    _sequencePosition++;
                //    _patternPosition++;
                //    return true;
                //}
                ////////
```

```csharp
                    //if (_filterPosition == _patternSequence.Length)
                    //{
                    //    _filterPosition = -2; // Длиннее чем нужно
                    //    return false;
                    //}
                    //if (element != _patternSequence[_filterPosition])
                    //{
                    //    _filterPosition = -1;
                    //    return false; // Начинается иначе
                    //}
                    //_filterPosition++;
                    //if (_filterPosition == (_patternSequence.Length - 1))
                    //    return false;
                    //if (_filterPosition >= 0)
                    //{
                    //    if (element == _patternSequence[_filterPosition + 1])
                    //        _filterPosition++;
                    //    else
                    //        return false;
                    //}
                    //if (_filterPosition < 0)
                    //{
                    //    if (element == _patternSequence[0])
                    //        _filterPosition = 0;
                    //}
                }

                public void AddAllPatternMatchedToResults(IEnumerable<ulong> sequencesToMatch)
                {
                    foreach (var sequenceToMatch in sequencesToMatch)
                    {
                        if (PatternMatch(sequenceToMatch))
                        {
                            _results.Add(sequenceToMatch);
                        }
                    }
                }
            }

            #endregion
        }
    }
```

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

```csharp
using Platform.Collections.Lists;
using Platform.Data.Sequences;
using System.Collections.Generic;

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

namespace Platform.Data.Doublets.Sequences
{
    public static class SequencesExtensions
    {
        public static TLink Create<TLink>(this ISequences<TLink> sequences, IList<TLink[]> groupedSequence)
        {
            var finalSequence = new TLink[groupedSequence.Count];
            for (var i = 0; i < finalSequence.Length; i++)
            {
                var part = groupedSequence[i];
                finalSequence[i] = part.Length == 1 ? part[0] : sequences.Create(part);
            }
            return sequences.Create(finalSequence);
        }

        public static IList<TLink> ToList<TLink>(this ISequences<TLink> sequences, TLink sequence)
        {
            var list = new List<TLink>();
            sequences.EachPart(list.AddAndReturnTrue, sequence);
            return list;
        }
    }
}
```

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

```csharp
using System;
using System.Collections.Generic;
```

```csharp
using Platform.Interfaces;
using Platform.Collections.Stacks;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Sequences.CreteriaMatchers;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Sequences.Indexes;

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

namespace Platform.Data.Doublets.Sequences
{
    public class SequencesOptions<TLink> // TODO: To use type parameter <TLink> the
    //     ILinks<TLink> must contain GetConstants function.
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        //     EqualityComparer<TLink>.Default;

        public TLink SequenceMarkerLink { get; set; }
        public bool UseCascadeUpdate { get; set; }
        public bool UseCascadeDelete { get; set; }
        public bool UseIndex { get; set; } // TODO: Update Index on sequence update/delete.
        public bool UseSequenceMarker { get; set; }
        public bool UseCompression { get; set; }
        public bool UseGarbageCollection { get; set; }
        public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting { get; set; }
        public bool EnforceSingleSequenceVersionOnWriteBasedOnNew { get; set; }

        public MarkedSequenceCriterionMatcher<TLink> MarkedSequenceMatcher { get; set; }
        public IConverter<IList<TLink>, TLink> LinksToSequenceConverter { get; set; }
        public ISequenceIndex<TLink> Index { get; set; }
        public ISequenceWalker<TLink> Walker { get; set; }

        // TODO: Реализовать компактификацию при чтении
        //public bool EnforceSingleSequenceVersionOnRead { get; set; }
        //public bool UseRequestMarker { get; set; }
        //public bool StoreRequestResults { get; set; }

        public void InitOptions(ISynchronizedLinks<TLink> links)
        {
            if (UseSequenceMarker)
            {
                if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
                {
                    SequenceMarkerLink = links.CreatePoint();
                }
                else
                {
                    if (!links.Exists(SequenceMarkerLink))
                    {
                        var link = links.CreatePoint();
                        if (!_equalityComparer.Equals(link, SequenceMarkerLink))
                        {
                            throw new InvalidOperationException("Cannot recreate sequence marker
                            //     link.");
                        }
                    }
                }
                if (MarkedSequenceMatcher == null)
                {
                    MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
                    //     SequenceMarkerLink);
                }
            }
            var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
            if (UseCompression)
            {
                if (LinksToSequenceConverter == null)
                {
                    ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
                    if (UseSequenceMarker)
                    {
                        totalSequenceSymbolFrequencyCounter = new
                        //     TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
                        //     MarkedSequenceMatcher);
                    }
                    else
                    {
```

```
76                         totalSequenceSymbolFrequencyCounter = new
                         ↪   TotalSequenceSymbolFrequencyCounter<TLink>(links);
77                     }
78                     var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
                         ↪   totalSequenceSymbolFrequencyCounter);
79                     var compressingConverter = new CompressingConverter<TLink>(links,
                         ↪   balancedVariantConverter, doubletFrequenciesCache);
80                     LinksToSequenceConverter = compressingConverter;
81                 }
82             }
83             else
84             {
85                 if (LinksToSequenceConverter == null)
86                 {
87                     LinksToSequenceConverter = balancedVariantConverter;
88                 }
89             }
90             if (UseIndex && Index == null)
91             {
92                 Index = new SequenceIndex<TLink>(links);
93             }
94             if (Walker == null)
95             {
96                 Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
97             }
98         }
99
100        public void ValidateOptions()
101        {
102            if (UseGarbageCollection && !UseSequenceMarker)
103            {
104                throw new NotSupportedException("To use garbage collection UseSequenceMarker
                 ↪   option must be on.");
105            }
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.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Collections.Stacks;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.Sequences.Walkers
8   {
9       public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
10      {
11          public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links, stack)
             ↪   { }
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          protected override TLink GetNextElementAfterPop(TLink element) =>
             ↪   Links.GetSource(element);
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          protected override TLink GetNextElementAfterPush(TLink element) =>
             ↪   Links.GetTarget(element);
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          protected override IEnumerable<TLink> WalkContents(TLink element)
21          {
22              var parts = Links.GetLink(element);
23              var start = Links.Constants.IndexPart + 1;
```

```
24              for (var i = parts.Count - 1; i >= start; i--)
25              {
26                  var part = parts[i];
27                  if (IsElement(part))
28                  {
29                      yield return part;
30                  }
31              }
32          }
33      }
34  }
```

./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 =
            ↪  EqualityComparer<TLink>.Default;
17
18          private readonly Func<TLink, bool> _isElement;
19
20          public LeveledSequenceWalker(ILinks<TLink> links, Func<TLink, bool> isElement) :
            ↪  base(links) => _isElement = isElement;
21
22          public LeveledSequenceWalker(ILinks<TLink> links) : base(links) => _isElement =
            ↪  Links.IsPartialPoint;
23
24          public IEnumerable<TLink> Walk(TLink sequence) => ToArray(sequence);
25
26          public TLink[] ToArray(TLink sequence)
27          {
28              var length = 1;
29              var array = new TLink[length];
30              array[0] = sequence;
31              if (_isElement(sequence))
32              {
33                  return array;
34              }
35              bool hasElements;
36              do
37              {
38                  length *= 2;
39  #if USEARRAYPOOL
40                  var nextArray = ArrayPool.Allocate<ulong>(length);
41  #else
42                  var nextArray = new TLink[length];
43  #endif
44                  hasElements = false;
45                  for (var i = 0; i < array.Length; i++)
46                  {
47                      var candidate = array[i];
48                      if (_equalityComparer.Equals(array[i], default))
49                      {
50                          continue;
51                      }
52                      var doubletOffset = i * 2;
53                      if (_isElement(candidate))
54                      {
55                          nextArray[doubletOffset] = candidate;
56                      }
57                      else
58                      {
59                          var link = Links.GetLink(candidate);
60                          var linkSource = Links.GetSource(link);
61                          var linkTarget = Links.GetTarget(link);
62                          nextArray[doubletOffset] = linkSource;
63                          nextArray[doubletOffset + 1] = linkTarget;
64                          if (!hasElements)
```

```csharp
                        {
                            hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
                        }
                    }
                }
#if USEARRAYPOOL
                if (array.Length > 1)
                {
                    ArrayPool.Free(array);
                }
#endif
                array = nextArray;
            }
            while (hasElements);
            var filledElementsCount = CountFilledElements(array);
            if (filledElementsCount == array.Length)
            {
                return array;
            }
            else
            {
                return CopyFilledElements(array, filledElementsCount);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLink[] CopyFilledElements(TLink[] array, int filledElementsCount)
        {
            var finalArray = new TLink[filledElementsCount];
            for (int i = 0, j = 0; i < array.Length; i++)
            {
                if (!_equalityComparer.Equals(array[i], default))
                {
                    finalArray[j] = array[i];
                    j++;
                }
            }
#if USEARRAYPOOL
            ArrayPool.Free(array);
#endif
            return finalArray;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static int CountFilledElements(TLink[] array)
        {
            var count = 0;
            for (var i = 0; i < array.Length; i++)
            {
                if (!_equalityComparer.Equals(array[i], default))
                {
                    count++;
                }
            }
            return count;
        }
    }
}
```

./Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs
```csharp
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections.Stacks;

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

namespace Platform.Data.Doublets.Sequences.Walkers
{
    public class RightSequenceWalker<TLink> : SequenceWalkerBase<TLink>
    {
        public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links,
        ↪  stack) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetNextElementAfterPop(TLink element) =>
        ↪  Links.GetTarget(element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetNextElementAfterPush(TLink element) =>
        ↪  Links.GetSource(element);
```

```
18
19                [MethodImpl(MethodImplOptions.AggressiveInlining)]
20                protected override IEnumerable<TLink> WalkContents(TLink element)
21                {
22                    var parts = Links.GetLink(element);
23                    for (var i = Links.Constants.IndexPart + 1; i < parts.Count; i++)
24                    {
25                        var part = parts[i];
26                        if (IsElement(part))
27                        {
28                            yield return part;
29                        }
30                    }
31                }
32            }
33        }
```

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

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

```
60        }
61    }
```

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

```csharp
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 =
             ↪  EqualityComparer<TLink>.Default;
11
12           private readonly ILinks<TLink> _links;
13           private readonly TLink _stack;
14
15           public bool IsEmpty => _equalityComparer.Equals(Peek(), _stack);
16
17           public Stack(ILinks<TLink> links, TLink stack)
18           {
19               _links = links;
20               _stack = stack;
21           }
22
23           private TLink GetStackMarker() => _links.GetSource(_stack);
24
25           private TLink GetTop() => _links.GetTarget(_stack);
26
27           public TLink Peek() => _links.GetTarget(GetTop());
28
29           public TLink Pop()
30           {
31               var element = Peek();
32               if (!_equalityComparer.Equals(element, _stack))
33               {
34                   var top = GetTop();
35                   var previousTop = _links.GetSource(top);
36                   _links.Update(_stack, GetStackMarker(), previousTop);
37                   _links.Delete(top);
38               }
39               return element;
40           }
41
42           public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
             ↪  _links.GetOrCreate(GetTop(), element));
43       }
44   }
```

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

```csharp
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

```csharp
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).
```

```csharp
        /// TODO: Try to unfold code of each method using IL generation for performance improvements.
        /// TODO: Or even to unfold multiple layers of implementations.
        /// </remarks>
        public class SynchronizedLinks<T> : ISynchronizedLinks<T>
        {
            public LinksCombinedConstants<T, T, int> Constants { get; }
            public ISynchronization SyncRoot { get; }
            public ILinks<T> Sync { get; }
            public ILinks<T> Unsync { get; }

            public SynchronizedLinks(ILinks<T> links) : this(new ReaderWriterLockSynchronization(),
                links) { }

            public SynchronizedLinks(ISynchronization synchronization, ILinks<T> links)
            {
                SyncRoot = synchronization;
                Sync = this;
                Unsync = links;
                Constants = links.Constants;
            }

            public T Count(IList<T> restriction) => SyncRoot.ExecuteReadOperation(restriction,
                Unsync.Count);
            public T Each(Func<IList<T>, T> handler, IList<T> restrictions) =>
                SyncRoot.ExecuteReadOperation(handler, restrictions, (handler1, restrictions1) =>
                Unsync.Each(handler1, restrictions1));
            public T Create() => SyncRoot.ExecuteWriteOperation(Unsync.Create);
            public T Update(IList<T> restrictions) => SyncRoot.ExecuteWriteOperation(restrictions,
                Unsync.Update);
            public void Delete(T link) => SyncRoot.ExecuteWriteOperation(link, Unsync.Delete);

            //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
            //  IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
            //{
            //    if (restriction != null && substitution != null &&
            //  !substitution.EqualTo(restriction))
            //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
            //  substitution, substitutedHandler, Unsync.Trigger);

            //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
            //  substitutedHandler, Unsync.Trigger);
            //}
        }
    }
```

./Platform.Data.Doublets/UInt64Link.cs

```csharp
using System;
using System.Collections;
using System.Collections.Generic;
using Platform.Exceptions;
using Platform.Ranges;
using Platform.Singletons;
using Platform.Collections.Lists;
using Platform.Data.Constants;

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

namespace Platform.Data.Doublets
{
    /// <summary>
    /// Структура описывающая уникальную связь.
    /// </summary>
    public struct UInt64Link : IEquatable<UInt64Link>, IReadOnlyList<ulong>, IList<ulong>
    {
        private static readonly LinksCombinedConstants<bool, ulong, int> _constants =
            Default<LinksCombinedConstants<bool, ulong, int>>.Instance;

        private const int Length = 3;

        public readonly ulong Index;
        public readonly ulong Source;
        public readonly ulong Target;

        public static readonly UInt64Link Null = new UInt64Link();

        public UInt64Link(params ulong[] values)
        {
            Index = values.Length > _constants.IndexPart ? values[_constants.IndexPart] :
                _constants.Null;
```

```csharp
                Source = values.Length > _constants.SourcePart ? values[_constants.SourcePart] :
                →   _constants.Null;
                Target = values.Length > _constants.TargetPart ? values[_constants.TargetPart] :
                →   _constants.Null;
        }

        public UInt64Link(IList<ulong> values)
        {
                Index = values.Count > _constants.IndexPart ? values[_constants.IndexPart] :
                →   _constants.Null;
                Source = values.Count > _constants.SourcePart ? values[_constants.SourcePart] :
                →   _constants.Null;
                Target = values.Count > _constants.TargetPart ? values[_constants.TargetPart] :
                →   _constants.Null;
        }

        public UInt64Link(ulong index, ulong source, ulong target)
        {
                Index = index;
                Source = source;
                Target = target;
        }

        public UInt64Link(ulong source, ulong target)
                : this(_constants.Null, source, target)
        {
                Source = source;
                Target = target;
        }

        public static UInt64Link Create(ulong source, ulong target) => new UInt64Link(source,
        →   target);

        public override int GetHashCode() => (Index, Source, Target).GetHashCode();

        public bool IsNull() => Index == _constants.Null
                              && Source == _constants.Null
                              && Target == _constants.Null;

        public override bool Equals(object other) => other is UInt64Link &&
        →   Equals((UInt64Link)other);

        public bool Equals(UInt64Link other) => Index == other.Index
                                              && Source == other.Source
                                              && Target == other.Target;

        public static string ToString(ulong index, ulong source, ulong target) => $"({index}:
        →   {source}->{target})";

        public static string ToString(ulong source, ulong target) => $"({source}->{target})";

        public static implicit operator ulong[](UInt64Link link) => link.ToArray();

        public static implicit operator UInt64Link(ulong[] linkArray) => new
        →   UInt64Link(linkArray);

        public override string ToString() => Index == _constants.Null ? ToString(Source, Target)
        →   : ToString(Index, Source, Target);

        #region IList

        public ulong this[int index]
        {
                get
                {
                        Ensure.Always.ArgumentInRange(index, new Range<int>(0, Length - 1),
                        →   nameof(index));
                        if (index == _constants.IndexPart)
                        {
                                return Index;
                        }
                        if (index == _constants.SourcePart)
                        {
                                return Source;
                        }
                        if (index == _constants.TargetPart)
                        {
                                return Target;
                        }
```

```
100                    throw new NotSupportedException(); // Impossible path due to
                       ↪  Ensure.ArgumentInRange
101                }
102            set => throw new NotSupportedException();
103        }

105        public int Count => Length;

107        public bool IsReadOnly => true;

109        IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

111        public IEnumerator<ulong> GetEnumerator()
112        {
113            yield return Index;
114            yield return Source;
115            yield return Target;
116        }

118        public void Add(ulong item) => throw new NotSupportedException();

120        public void Clear() => throw new NotSupportedException();

122        public bool Contains(ulong item) => IndexOf(item) >= 0;

124        public void CopyTo(ulong[] array, int arrayIndex)
125        {
126            Ensure.Always.ArgumentNotNull(array, nameof(array));
127            Ensure.Always.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
                   ↪  nameof(arrayIndex));
128            if (arrayIndex + Length > array.Length)
129            {
130                throw new ArgumentException();
131            }
132            array[arrayIndex++] = Index;
133            array[arrayIndex++] = Source;
134            array[arrayIndex] = Target;
135        }

137        public bool Remove(ulong item) => Throw.A.NotSupportedExceptionAndReturn<bool>();

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

154            return -1;
155        }

157        public void Insert(int index, ulong item) => throw new NotSupportedException();

159        public void RemoveAt(int index) => throw new NotSupportedException();

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) =>
               ↪  Point<ulong>.IsPartialPoint(link);
9      }
10 }
```

```csharp
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 =
                Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
16
17          public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
18
19          public static void EnsureEachLinkExists(this ILinks<ulong> links, IList<ulong> sequence)
20          {
21              if (sequence == null)
22              {
23                  return;
24              }
25              for (var i = 0; i < sequence.Count; i++)
26              {
27                  if (!links.Exists(sequence[i]))
28                  {
29                      throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                            $"sequence[{i}]");
30                  }
31              }
32          }
33
34          public static void EnsureEachLinkIsAnyOrExists(this ILinks<ulong> links, IList<ulong>
                sequence)
35          {
36              if (sequence == null)
37              {
38                  return;
39              }
40              for (var i = 0; i < sequence.Count; i++)
41              {
42                  if (sequence[i] != Constants.Any && !links.Exists(sequence[i]))
43                  {
44                      throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                            $"sequence[{i}]");
45                  }
46              }
47          }
48
49          public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
50          {
51              if (sequence == null)
52              {
53                  return false;
54              }
55              var constants = links.Constants;
56              for (var i = 0; i < sequence.Length; i++)
57              {
58                  if (sequence[i] == constants.Any)
59                  {
60                      return true;
61                  }
62              }
63              return false;
64          }
65
66          public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
                Func<UInt64Link, bool> isElement, bool renderIndex = false, bool renderDebug = false)
67          {
68              var sb = new StringBuilder();
69              var visited = new HashSet<ulong>();
70              links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
                    innerSb.Append(link.Index), renderIndex, renderDebug);
71              return sb.ToString();
72          }
```

```
73
74        public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
      ↪  Func<UInt64Link, bool> isElement, Action<StringBuilder, UInt64Link> appendElement,
      ↪  bool renderIndex = false, bool renderDebug = false)
75        {
76            var sb = new StringBuilder();
77            var visited = new HashSet<ulong>();
78            links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
      ↪  renderDebug);
79            return sb.ToString();
80        }
81
82        public static void AppendStructure(this ILinks<ulong> links, StringBuilder sb,
      ↪  HashSet<ulong> visited, ulong linkIndex, Func<UInt64Link, bool> isElement,
      ↪  Action<StringBuilder, UInt64Link> appendElement, bool renderIndex = false, bool
      ↪  renderDebug = false)
83        {
84            if (sb == null)
85            {
86                throw new ArgumentNullException(nameof(sb));
87            }
88            if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
      ↪  Constants.Itself)
89            {
90                return;
91            }
92            if (links.Exists(linkIndex))
93            {
94                if (visited.Add(linkIndex))
95                {
96                    sb.Append('(');
97                    var link = new UInt64Link(links.GetLink(linkIndex));
98                    if (renderIndex)
99                    {
100                       sb.Append(link.Index);
101                       sb.Append(':');
102                   }
103                   if (link.Source == link.Index)
104                   {
105                       sb.Append(link.Index);
106                   }
107                   else
108                   {
109                       var source = new UInt64Link(links.GetLink(link.Source));
110                       if (isElement(source))
111                       {
112                           appendElement(sb, source);
113                       }
114                       else
115                       {
116                           links.AppendStructure(sb, visited, source.Index, isElement,
      ↪  appendElement, renderIndex);
117                       }
118                   }
119                   sb.Append(' ');
120                   if (link.Target == link.Index)
121                   {
122                       sb.Append(link.Index);
123                   }
124                   else
125                   {
126                       var target = new UInt64Link(links.GetLink(link.Target));
127                       if (isElement(target))
128                       {
129                           appendElement(sb, target);
130                       }
131                       else
132                       {
133                           links.AppendStructure(sb, visited, target.Index, isElement,
      ↪  appendElement, renderIndex);
134                       }
135                   }
136                   sb.Append(')');
137               }
138               else
139               {
140                   if (renderDebug)
141                   {
```

```csharp
                            sb.Append('*');
                        }
                        sb.Append(linkIndex);
                    }
                }
                else
                {
                    if (renderDebug)
                    {
                        sb.Append('~');
                    }
                    sb.Append(linkIndex);
                }
            }
        }
    }
}
```

./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs

```csharp
using System;
using System.Linq;
using System.Collections.Generic;
using System.IO;
using System.Runtime.CompilerServices;
using System.Threading;
using System.Threading.Tasks;
using Platform.Disposables;
using Platform.Timestamps;
using Platform.Unsafe;
using Platform.IO;
using Platform.Data.Doublets.Decorators;

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

namespace Platform.Data.Doublets
{
    public class UInt64LinksTransactionsLayer : LinksDisposableDecoratorBase<ulong> //-V3073
    {
        /// <remarks>
        /// Альтернативные варианты хранения трансформации (элемента транзакции):
        ///
        /// private enum TransitionType
        /// {
        ///     Creation,
        ///     UpdateOf,
        ///     UpdateTo,
        ///     Deletion
        /// }
        ///
        /// private struct Transition
        /// {
        ///     public ulong TransactionId;
        ///     public UniqueTimestamp Timestamp;
        ///     public TransactionItemType Type;
        ///     public Link Source;
        ///     public Link Linker;
        ///     public Link Target;
        /// }
        ///
        /// Или
        ///
        /// public struct TransitionHeader
        /// {
        ///     public ulong TransactionIdCombined;
        ///     public ulong TimestampCombined;
        ///
        ///     public ulong TransactionId
        ///     {
        ///         get
        ///         {
        ///             return (ulong) mask &amp; TransactionIdCombined;
        ///         }
        ///     }
        ///
        ///     public UniqueTimestamp Timestamp
        ///     {
        ///         get
        ///         {
        ///             return (UniqueTimestamp)mask &amp; TransactionIdCombined;
        ///         }
```

```
 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
           ↪ transactionId, UInt64Link before, UInt64Link after)
 94        {
 95            TransactionId = transactionId;
 96            Before = before;
 97            After = after;
 98            Timestamp = uniqueTimestampFactory.Create();
 99        }
100
101        public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
           ↪ transactionId, UInt64Link before)
102            : this(uniqueTimestampFactory, transactionId, before, default)
103        {
104        }
105
106        public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId)
107            : this(uniqueTimestampFactory, transactionId, default, default)
108        {
109        }
110
111        public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
           ↪ {After}";
112    }
113
114    /// <remarks>
115    /// Другие варианты реализации транзакций (атомарности):
116    ///     1. Разделение хранения значения связи ((Source Target) или (Source Linker
       ↪ Target)) и индексов.
117    ///     2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
       ↪ потребуется решить вопрос
118    ///        со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
       ↪ пересечениями идентификаторов.
119    ///
120    /// Где хранить промежуточный список транзакций?
121    ///
122    /// В оперативной памяти:
123    ///   Минусы:
124    ///     1. Может усложнить систему, если она будет функционировать самостоятельно,
125    ///     так как нужно отдельно выделять память под список трансформаций.
126    ///     2. Выделенной оперативной памяти может не хватить, в том случае,
127    ///     если транзакция использует слишком много трансформаций.
128    ///        -> Можно использовать жёсткий диск для слишком длинных транзакций.
129    ///        -> Максимальный размер списка трансформаций можно ограничить / задать
       ↪ константой.
130    ///     3. При подтверждении транзакции (Commit) все трансформации записываются разом
       ↪ создавая задержку.
131    ///
```

```csharp
        /// На жёстком диске:
        ///   Минусы:
        ///     1. Длительный отклик, на запись каждой трансформации.
        ///     2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
        ///         -> Это может решаться упаковкой/исключением дублирующих операций.
        ///         -> Также это может решаться тем, что короткие транзакции вообще
        ///            не будут записываться в случае отката.
        ///     3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
        ///  операции (трансформации)
        ///         будут записаны в лог.
        ///
        /// </remarks>
        public class Transaction : DisposableBase
        {
            private readonly Queue<Transition> _transitions;
            private readonly UInt64LinksTransactionsLayer _layer;
            public bool IsCommitted { get; private set; }
            public bool IsReverted { get; private set; }

            public Transaction(UInt64LinksTransactionsLayer layer)
            {
                _layer = layer;
                if (_layer._currentTransactionId != 0)
                {
                    throw new NotSupportedException("Nested transactions not supported.");
                }
                IsCommitted = false;
                IsReverted = false;
                _transitions = new Queue<Transition>();
                SetCurrentTransaction(layer, this);
            }

            public void Commit()
            {
                EnsureTransactionAllowsWriteOperations(this);
                while (_transitions.Count > 0)
                {
                    var transition = _transitions.Dequeue();
                    _layer._transitions.Enqueue(transition);
                }
                _layer._lastCommitedTransactionId = _layer._currentTransactionId;
                IsCommitted = true;
            }

            private void Revert()
            {
                EnsureTransactionAllowsWriteOperations(this);
                var transitionsToRevert = new Transition[_transitions.Count];
                _transitions.CopyTo(transitionsToRevert, 0);
                for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
                {
                    _layer.RevertTransition(transitionsToRevert[i]);
                }
                IsReverted = true;
            }

            public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
             Transaction transaction)
            {
                layer._currentTransactionId = layer._lastCommitedTransactionId + 1;
                layer._currentTransactionTransitions = transaction._transitions;
                layer._currentTransaction = transaction;
            }

            public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
            {
                if (transaction.IsReverted)
                {
                    throw new InvalidOperationException("Transation is reverted.");
                }
                if (transaction.IsCommitted)
                {
                    throw new InvalidOperationException("Transation is commited.");
                }
            }

            protected override void Dispose(bool manual, bool wasDisposed)
            {
                if (!wasDisposed && _layer != null && !_layer.IsDisposed)
```

```
209                 {
210                     if (!IsCommitted && !IsReverted)
211                     {
212                         Revert();
213                     }
214                     _layer.ResetCurrentTransation();
215                 }
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 _lastCommitedTransition;
227         private ulong _currentTransactionId;
228         private Queue<Transition> _currentTransactionTransitions;
229         private Transaction _currentTransaction;
230         private ulong _lastCommitedTransactionId;
231
232         public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
233             : base(links)
234         {
235             if (string.IsNullOrWhiteSpace(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 lastCommitedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
244             var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
245             if (!lastCommitedTransition.Equals(lastWrittenTransition))
246             {
247                 Dispose();
248                 throw new NotSupportedException("Database is damaged, autorecovery is not
                    ↪  supported yet.");
249             }
250             if (lastCommitedTransition.Equals(default(Transition)))
251             {
252                 FileHelpers.WriteFirst(logAddress, lastCommitedTransition);
253             }
254             _lastCommitedTransition = lastCommitedTransition;
255             // TODO: Think about a better way to calculate or store this value
256             var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
257             _lastCommitedTransactionId = allTransitions.Max(x => x.TransactionId);
258             _uniqueTimestampFactory = new UniqueTimestampFactory();
259             _logAddress = logAddress;
260             _log = FileHelpers.Append(logAddress);
261             _transitions = new Queue<Transition>();
262             _transitionsPusher = new Task(TransitionsPusher);
263             _transitionsPusher.Start();
264         }
265
266         public IList<ulong> GetLinkValue(ulong link) => Links.GetLink(link);
267
268         public override ulong Create()
269         {
270             var createdLinkIndex = Links.Create();
271             var createdLink = new UInt64Link(Links.GetLink(createdLinkIndex));
272             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                ↪  default, createdLink));
273             return createdLinkIndex;
274         }
275
276         public override ulong Update(IList<ulong> parts)
277         {
278             var linkIndex = parts[Constants.IndexPart];
279             var beforeLink = new UInt64Link(Links.GetLink(linkIndex));
280             linkIndex = Links.Update(parts);
281             var afterLink = new UInt64Link(Links.GetLink(linkIndex));
282             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                ↪  beforeLink, afterLink));
283             return linkIndex;
284         }
```

```csharp
        public override void Delete(ulong link)
        {
            var deletedLink = new UInt64Link(Links.GetLink(link));
            Links.Delete(link);
            CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
            ↪   deletedLink, default));
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
        ↪   _transitions;

        private void CommitTransition(Transition transition)
        {
            if (_currentTransaction != null)
            {
                Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
            }
            var transitions = GetCurrentTransitions();
            transitions.Enqueue(transition);
        }

        private void RevertTransition(Transition transition)
        {
            if (transition.After.IsNull()) // Revert Deletion with Creation
            {
                Links.Create();
            }
            else if (transition.Before.IsNull()) // Revert Creation with Deletion
            {
                Links.Delete(transition.After.Index);
            }
            else // Revert Update
            {
                Links.Update(new[] { transition.After.Index, transition.Before.Source,
                ↪   transition.Before.Target });
            }
        }

        private void ResetCurrentTransation()
        {
            _currentTransactionId = 0;
            _currentTransactionTransitions = null;
            _currentTransaction = null;
        }

        private void PushTransitions()
        {
            if (_log == null || _transitions == null)
            {
                return;
            }
            for (var i = 0; i < _transitions.Count; i++)
            {
                var transition = _transitions.Dequeue();

                _log.Write(transition);
                _lastCommitedTransition = transition;
            }
        }

        private void TransitionsPusher()
        {
            while (!IsDisposed && _transitionsPusher != null)
            {
                Thread.Sleep(DefaultPushDelay);
                PushTransitions();
            }
        }

        public Transaction BeginTransaction() => new Transaction(this);

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

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

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

```
17              {
18                  _charToUnicodeSymbolConverter = charToUnicodeSymbolConverter;
19                  _index = index;
20                  _listToSequenceLinkConverter = listToSequenceLinkConverter;
21                  _unicodeSequenceMarker = unicodeSequenceMarker;
22              }
23
24              public TLink Convert(string source)
25              {
26                  var elements = new 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
                        ↪ amount of NIL characters before actual Character)
47                      var createdLink = _links.CreatePoint();
48                      _links.Update(createdLink, firstLink, createdLink);
49                      if (createdLink != i)
50                      {
51                          throw new InvalidOperationException("Unable to initialize UTF 16
                            ↪ table.");
52                      }
53                  }
54              }
55          }
56
```

```csharp
            // 0 - null link
            // 1 - nil character (0 character)
            // ...
            // 65536 (0(1) + 65535 = 65536 possible values)

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static ulong FromCharToLink(char character) => (ulong)character + 1;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static char FromLinkToChar(ulong link) => (char)(link - 1);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static bool IsCharLink(ulong link) => link <= MapSize;

            public static string FromLinksToString(IList<ulong> linksList)
            {
                var sb = new StringBuilder();
                for (int i = 0; i < linksList.Count; i++)
                {
                    sb.Append(FromLinkToChar(linksList[i]));
                }
                return sb.ToString();
            }

            public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
            {
                var sb = new StringBuilder();
                if (links.Exists(link))
                {
                    StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
                        x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
                        element =>
                        {
                            sb.Append(FromLinkToChar(element));
                            return true;
                        });
                }
                return sb.ToString();
            }

            public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
                chars.Length);

            public static ulong[] FromCharsToLinkArray(char[] chars, int count)
            {
                // char array to ulong array
                var linksSequence = new ulong[count];
                for (var i = 0; i < count; i++)
                {
                    linksSequence[i] = FromCharToLink(chars[i]);
                }
                return linksSequence;
            }

            public static ulong[] FromStringToLinkArray(string sequence)
            {
                // char array to ulong array
                var linksSequence = new ulong[sequence.Length];
                for (var i = 0; i < sequence.Length; i++)
                {
                    linksSequence[i] = FromCharToLink(sequence[i]);
                }
                return linksSequence;
            }

            public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
            {
                var result = new List<ulong[]>();
                var offset = 0;
                while (offset < sequence.Length)
                {
                    var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
                    var relativeLength = 1;
                    var absoluteLength = offset + relativeLength;
                    while (absoluteLength < sequence.Length &&
                            currentCategory ==
                            CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
                    {
                        relativeLength++;
```

```csharp
                            absoluteLength++;
                        }
                        // char array to ulong array
                        var innerSequence = new ulong[relativeLength];
                        var maxLength = offset + relativeLength;
                        for (var i = offset; i < maxLength; i++)
                        {
                            innerSequence[i - offset] = FromCharToLink(sequence[i]);
                        }
                        result.Add(innerSequence);
                        offset += relativeLength;
                    }
                    return result;
                }

                public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
                {
                    var result = new List<ulong[]>();
                    var offset = 0;
                    while (offset < array.Length)
                    {
                        var relativeLength = 1;
                        if (array[offset] <= LastCharLink)
                        {
                            var currentCategory =
                            ↪  CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
                            var absoluteLength = offset + relativeLength;
                            while (absoluteLength < array.Length &&
                                    array[absoluteLength] <= LastCharLink &&
                                    currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(⌋
                                    ↪  array[absoluteLength])))
                            {
                                relativeLength++;
                                absoluteLength++;
                            }
                        }
                        else
                        {
                            var absoluteLength = offset + relativeLength;
                            while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
                            {
                                relativeLength++;
                                absoluteLength++;
                            }
                        }
                        // copy array
                        var innerSequence = new ulong[relativeLength];
                        var maxLength = offset + relativeLength;
                        for (var i = offset; i < maxLength; i++)
                        {
                            innerSequence[i - offset] = array[i];
                        }
                        result.Add(innerSequence);
                        offset += relativeLength;
                    }
                    return result;
                }
            }
        }
```

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

```csharp
using Platform.Interfaces;
using System.Collections.Generic;

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

namespace Platform.Data.Doublets.Unicode
{
    public class UnicodeSequenceCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
    ↪  ICriterionMatcher<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
        private readonly TLink _unicodeSequenceMarker;
        public UnicodeSequenceCriterionMatcher(ILinks<TLink> links, TLink unicodeSequenceMarker)
        ↪   : base(links) => _unicodeSequenceMarker = unicodeSequenceMarker;
        public bool IsMatched(TLink link) => _equalityComparer.Equals(Links.GetTarget(link),
        ↪  _unicodeSequenceMarker);
    }
```

```
15    }

./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>,
         ↪   IConverter<TLink, string>
11       {
12          private readonly ICriterionMatcher<TLink> _unicodeSequenceCriterionMatcher;
13          private readonly ISequenceWalker<TLink> _sequenceWalker;
14          private readonly IConverter<TLink, char> _unicodeSymbolToCharConverter;
15
16          public UnicodeSequenceToStringConverter(ILinks<TLink> links, ICriterionMatcher<TLink>
            ↪   unicodeSequenceCriterionMatcher, ISequenceWalker<TLink> sequenceWalker,
            ↪   IConverter<TLink, char> unicodeSymbolToCharConverter) : base(links)
17          {
18             _unicodeSequenceCriterionMatcher = unicodeSequenceCriterionMatcher;
19             _sequenceWalker = sequenceWalker;
20             _unicodeSymbolToCharConverter = unicodeSymbolToCharConverter;
21          }
22
23          public string Convert(TLink source)
24          {
25             if(!_unicodeSequenceCriterionMatcher.IsMatched(source))
26             {
27                throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
                  ↪   not a unicode sequence.");
28             }
29             var sequence = Links.GetSource(source);
30             var charArray = _sequenceWalker.Walk(sequence).Select(_unicodeSymbolToCharConverter.↵
               ↪   Convert).ToArray();
31             return new string(charArray);
32          }
33       }
34    }

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

./Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs
1    using Platform.Interfaces;
2    using Platform.Numbers;
3    using System;
4    using System.Collections.Generic;
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 UnicodeSymbolToCharConverter<TLink> : LinksOperatorBase<TLink>,
         ↪   IConverter<TLink, char>
11       {
12          private readonly IConverter<TLink> _unaryNumberToAddressConverter;
13          private readonly ICriterionMatcher<TLink> _unicodeSymbolCriterionMatcher;
```

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

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

```csharp
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 GreaterOrEqualPerfomanceTest()
19          {
20              const int N = 1000000;
21
22              ulong x = 10;
23              ulong y = 500;
24
25              bool result = false;
26
27              var ts1 = Performance.Measure(() =>
28              {
29                  for (int i = 0; i < N; i++)
30                  {
31                      result = Compare(x, y) >= 0;
32                  }
33              });
34
35              var comparer1 = Comparer<ulong>.Default;
36
37              var ts2 = Performance.Measure(() =>
38              {
39                  for (int i = 0; i < N; i++)
40                  {
41                      result = comparer1.Compare(x, y) >= 0;
42                  }
43              });
44
45              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/DoubletLinksTests.cs

```csharp
1  using System.Collections.Generic;
2  using Xunit;
3  using Platform.Reflection;
4  using Platform.Numbers;
5  using Platform.Memory;
6  using Platform.Scopes;
7  using Platform.Setters;
8  using Platform.Data.Doublets.ResizableDirectMemory;
9
10 namespace Platform.Data.Doublets.Tests
11 {
12     public static class DoubletLinksTests
13     {
14         [Fact]
15         public static void UInt64CRUDTest()
16         {
17             using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪ ResizableDirectMemoryLinks<ulong>>>())
18             {
19                 scope.Use<ILinks<ulong>>().TestCRUDOperations();
20             }
21         }
22
23         [Fact]
24         public static void UInt32CRUDTest()
25         {
26             using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪ ResizableDirectMemoryLinks<uint>>>())
27             {
28                 scope.Use<ILinks<uint>>().TestCRUDOperations();
29             }
30         }
31
32         [Fact]
33         public static void UInt16CRUDTest()
34         {
35             using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪ ResizableDirectMemoryLinks<ushort>>>())
36             {
37                 scope.Use<ILinks<ushort>>().TestCRUDOperations();
38             }
39         }
40
41         [Fact]
42         public static void UInt8CRUDTest()
43         {
44             using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪ ResizableDirectMemoryLinks<byte>>>())
45             {
46                 scope.Use<ILinks<byte>>().TestCRUDOperations();
47             }
48         }
49
50         private static void TestCRUDOperations<T>(this ILinks<T> links)
51         {
52             var constants = links.Constants;
53
54             var equalityComparer = EqualityComparer<T>.Default;
55
56             // Create Link
57             Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));
58
59             var setter = new Setter<T>(constants.Null);
60             links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
61
62             Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
```

```
63
64          var linkAddress = links.Create();
65
66          var link = new Link<T>(links.GetLink(linkAddress));
67
68          Assert.True(link.Count == 3);
69          Assert.True(equalityComparer.Equals(link.Index, linkAddress));
70          Assert.True(equalityComparer.Equals(link.Source, constants.Null));
71          Assert.True(equalityComparer.Equals(link.Target, constants.Null));
72
73          Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.One));
74
75          // Get first link
76          setter = new Setter<T>(constants.Null);
77          links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);
78
79          Assert.True(equalityComparer.Equals(setter.Result, linkAddress));
80
81          // Update link to reference itself
82          links.Update(linkAddress, linkAddress, linkAddress);
83
84          link = new Link<T>(links.GetLink(linkAddress));
85
86          Assert.True(equalityComparer.Equals(link.Source, linkAddress));
87          Assert.True(equalityComparer.Equals(link.Target, linkAddress));
88
89          // Update link to reference null (prepare for delete)
90          var updated = links.Update(linkAddress, constants.Null, constants.Null);
91
92          Assert.True(equalityComparer.Equals(updated, linkAddress));
93
94          link = new Link<T>(links.GetLink(linkAddress));
95
96          Assert.True(equalityComparer.Equals(link.Source, constants.Null));
97          Assert.True(equalityComparer.Equals(link.Target, constants.Null));
98
99          // Delete link
100         links.Delete(linkAddress);
101
102         Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));
103
104         setter = new Setter<T>(constants.Null);
105         links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
106
107         Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
108     }
109
110     [Fact]
111     public static void UInt64RawNumbersCRUDTest()
112     {
113         using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪ ResizableDirectMemoryLinks<ulong>>>())
114         {
115             scope.Use<ILinks<ulong>>().TestRawNumbersCRUDOperations();
116         }
117     }
118
119     [Fact]
120     public static void UInt32RawNumbersCRUDTest()
121     {
122         using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪ ResizableDirectMemoryLinks<uint>>>())
123         {
124             scope.Use<ILinks<uint>>().TestRawNumbersCRUDOperations();
125         }
126     }
127
128     [Fact]
129     public static void UInt16RawNumbersCRUDTest()
130     {
131         using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪ ResizableDirectMemoryLinks<ushort>>>())
132         {
133             scope.Use<ILinks<ushort>>().TestRawNumbersCRUDOperations();
134         }
135     }
136
137     [Fact]
138     public static void UInt8RawNumbersCRUDTest()
```

```
                {
                    using (var scope = new Scope<Types<HeapResizableDirectMemory,
                    ↪  ResizableDirectMemoryLinks<byte>>>())
                    {
                        scope.Use<ILinks<byte>>().TestRawNumbersCRUDOperations();
                    }
                }

        private static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
        {
            // Constants
            var constants = links.Constants;
            var equalityComparer = EqualityComparer<T>.Default;

            var h106E = new Hybrid<T>(106L, isExternal: true);
            var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
            var h108E = new Hybrid<T>(-108L);

            Assert.Equal(106L, h106E.AbsoluteValue);
            Assert.Equal(107L, h107E.AbsoluteValue);
            Assert.Equal(108L, h108E.AbsoluteValue);

            // Create Link (External -> External)
            var linkAddress1 = links.Create();

            links.Update(linkAddress1, h106E, h108E);

            var link1 = new Link<T>(links.GetLink(linkAddress1));

            Assert.True(equalityComparer.Equals(link1.Source, h106E));
            Assert.True(equalityComparer.Equals(link1.Target, h108E));

            // Create Link (Internal -> External)
            var linkAddress2 = links.Create();

            links.Update(linkAddress2, linkAddress1, h108E);

            var link2 = new Link<T>(links.GetLink(linkAddress2));

            Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
            Assert.True(equalityComparer.Equals(link2.Target, h108E));

            // Create Link (Internal -> Internal)
            var linkAddress3 = links.Create();

            links.Update(linkAddress3, linkAddress1, linkAddress2);

            var link3 = new Link<T>(links.GetLink(linkAddress3));

            Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
            Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));

            // Search for created link
            var setter1 = new Setter<T>(constants.Null);
            links.Each(h106E, h108E, setter1.SetAndReturnFalse);

            Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));

            // Search for nonexistent link
            var setter2 = new Setter<T>(constants.Null);
            links.Each(h106E, h107E, setter2.SetAndReturnFalse);

            Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));

            // Update link to reference null (prepare for delete)
            var updated = links.Update(linkAddress3, constants.Null, constants.Null);

            Assert.True(equalityComparer.Equals(updated, linkAddress3));

            link3 = new Link<T>(links.GetLink(linkAddress3));

            Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
            Assert.True(equalityComparer.Equals(link3.Target, constants.Null));

            // Delete link
            links.Delete(linkAddress3);

            Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Two));

            var setter3 = new Setter<T>(constants.Null);
```

```
218            links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);

219
220            Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
221        }

222
223        // TODO: Test layers
224    }
225 }
```

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

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

```
71              for (int i = 0; i < N; i++)
72              {
73                  result = equalityComparer2.Equals(x, y);
74              }
75          });

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

./Platform.Data.Doublets.Tests/LinksTests.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.IO;
5  using System.Text;
6  using System.Threading;
7  using System.Threading.Tasks;
8  using Xunit;
9  using Platform.Disposables;
10 using Platform.IO;
11 using Platform.Ranges;
12 using Platform.Random;
13 using Platform.Timestamps;
14 using Platform.Singletons;
15 using Platform.Counters;
16 using Platform.Diagnostics;
17 using Platform.Data.Constants;
18 using Platform.Data.Doublets.ResizableDirectMemory;
19 using Platform.Data.Doublets.Decorators;
20
21 namespace Platform.Data.Doublets.Tests
22 {
23     public static class LinksTests
24     {
25         private static readonly LinksCombinedConstants<bool, ulong, int> _constants =
            ↪  Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
26
27         private const long Iterations = 10 * 1024;
28
29         #region Concept
30
31         [Fact]
32         public static void MultipleCreateAndDeleteTest()
33         {
34             //const int N = 21;
35
36             using (var scope = new TempLinksTestScope())
37             {
38                 var links = scope.Links;
39
40                 for (var N = 0; N < 100; N++)
41                 {
42                     var random = new System.Random(N);
43
```

```csharp
                    var created = 0;
                    var deleted = 0;

                    for (var i = 0; i < N; i++)
                    {
                        var linksCount = links.Count();

                        var createPoint = random.NextBoolean();

                        if (linksCount > 2 && createPoint)
                        {
                            var linksAddressRange = new Range<ulong>(1, linksCount);
                            var source = random.NextUInt64(linksAddressRange);
                            var target = random.NextUInt64(linksAddressRange); //-V3086

                            var resultLink = links.CreateAndUpdate(source, target);
                            if (resultLink > linksCount)
                            {
                                created++;
                            }
                        }
                        else
                        {
                            links.Create();
                            created++;
                        }
                    }

                    Assert.True(created == (int)links.Count());

                    for (var i = 0; i < N; i++)
                    {
                        var link = (ulong)i + 1;
                        if (links.Exists(link))
                        {
                            links.Delete(link);
                            deleted++;
                        }
                    }

                    Assert.True(links.Count() == 0);
                }
            }
        }

        [Fact]
        public static void CascadeUpdateTest()
        {
            var itself = _constants.Itself;

            using (var scope = new TempLinksTestScope(useLog: true))
            {
                var links = scope.Links;

                var l1 = links.Create();
                var l2 = links.Create();

                l2 = links.Update(l2, l2, l1, l2);

                links.CreateAndUpdate(l2, itself);
                links.CreateAndUpdate(l2, itself);

                l2 = links.Update(l2, l1);

                links.Delete(l2);

                Global.Trash = links.Count();

                links.Unsync.DisposeIfPossible(); // Close links to access log

                Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop
                ↪  e.TempTransactionLogFilename);
            }
        }

        [Fact]
        public static void BasicTransactionLogTest()
        {
            using (var scope = new TempLinksTestScope(useLog: true))
            {
```

```csharp
123                    var links = scope.Links;
124                    var l1 = links.Create();
125                    var l2 = links.Create();
126
127                    Global.Trash = links.Update(l2, l2, l1, l2);
128
129                    links.Delete(l1);
130
131                    links.Unsync.DisposeIfPossible(); // Close links to access log
132
133                    Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop
     ↪      e.TempTransactionLogFilename);
134                }
135            }
136
137            [Fact]
138            public static void TransactionAutoRevertedTest()
139            {
140                // Auto Reverted (Because no commit at transaction)
141                using (var scope = new TempLinksTestScope(useLog: true))
142                {
143                    var links = scope.Links;
144                    var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
145                    using (var transaction = transactionsLayer.BeginTransaction())
146                    {
147                        var l1 = links.Create();
148                        var l2 = links.Create();
149
150                        links.Update(l2, l2, l1, l2);
151                    }
152
153                    Assert.Equal(0UL, links.Count());
154
155                    links.Unsync.DisposeIfPossible();
156
157                    var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s
     ↪      cope.TempTransactionLogFilename);
158                    Assert.Single(transitions);
159                }
160            }
161
162            [Fact]
163            public static void TransactionUserCodeErrorNoDataSavedTest()
164            {
165                // User Code Error (Autoreverted), no data saved
166                var itself = _constants.Itself;
167
168                TempLinksTestScope lastScope = null;
169                try
170                {
171                    using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
     ↪      useLog: true))
172                    {
173                        var links = scope.Links;
174                        var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor
     ↪      atorBase<ulong>)links.Unsync).Links;
175                        using (var transaction = transactionsLayer.BeginTransaction())
176                        {
177                            var l1 = links.CreateAndUpdate(itself, itself);
178                            var 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                            //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi
     ↪      tion>(scope.TempTransactionLogFilename);
186
187                            l2 = links.Update(l2, l1);
188
189                            links.Delete(l2);
190
191                            ExceptionThrower();
192
193                            transaction.Commit();
194                        }
195
196                        Global.Trash = links.Count();
```

```csharp
197                     }
198                 }
199             catch
200             {
201                 Assert.False(lastScope == null);
202
203                 var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l↵
        ↪   astScope.TempTransactionLogFilename);
204
205                 Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
        ↪   transitions[0].After.IsNull());
206
207                 lastScope.DeleteFiles();
208             }
209         }
210
211         [Fact]
212         public static void TransactionUserCodeErrorSomeDataSavedTest()
213         {
214             // User Code Error (Autoreverted), some data saved
215             var itself = _constants.Itself;
216
217             TempLinksTestScope lastScope = null;
218             try
219             {
220                 ulong l1;
221                 ulong l2;
222
223                 using (var scope = new TempLinksTestScope(useLog: true))
224                 {
225                     var links = scope.Links;
226                     l1 = links.CreateAndUpdate(itself, itself);
227                     l2 = links.CreateAndUpdate(itself, itself);
228
229                     l2 = links.Update(l2, l2, l1, l2);
230
231                     links.CreateAndUpdate(l2, itself);
232                     links.CreateAndUpdate(l2, itself);
233
234                     links.Unsync.DisposeIfPossible();
235
236                     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(↵
        ↪   scope.TempTransactionLogFilename);
237                 }
238
239                 using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
        ↪   useLog: true))
240                 {
241                     var links = scope.Links;
242                     var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
243                     using (var transaction = transactionsLayer.BeginTransaction())
244                     {
245                         l2 = links.Update(l2, l1);
246
247                         links.Delete(l2);
248
249                         ExceptionThrower();
250
251                         transaction.Commit();
252                     }
253
254                     Global.Trash = links.Count();
255                 }
256             }
257             catch
258             {
259                 Assert.False(lastScope == null);
260
261                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last↵
        ↪   Scope.TempTransactionLogFilename);
262
263                 lastScope.DeleteFiles();
264             }
265         }
266
267         [Fact]
268         public static void TransactionCommit()
269         {
270             var itself = _constants.Itself;
271
```

```
272            var tempDatabaseFilename = Path.GetTempFileName();
273            var tempTransactionLogFilename = Path.GetTempFileName();
274
275            // Commit
276            using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
       ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
       ↪ tempTransactionLogFilename))
277            using (var links = new UInt64Links(memoryAdapter))
278            {
279                using (var transaction = memoryAdapter.BeginTransaction())
280                {
281                    var l1 = links.CreateAndUpdate(itself, itself);
282                    var l2 = links.CreateAndUpdate(itself, itself);
283
284                    Global.Trash = links.Update(l2, l2, l1, l2);
285
286                    links.Delete(l1);
287
288                    transaction.Commit();
289                }
290
291                Global.Trash = links.Count();
292            }
293
294            Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran↵
       ↪ sactionLogFilename);
295        }
296
297        [Fact]
298        public static void TransactionDamage()
299        {
300            var itself = _constants.Itself;
301
302            var tempDatabaseFilename = Path.GetTempFileName();
303            var tempTransactionLogFilename = Path.GetTempFileName();
304
305            // Commit
306            using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
       ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
       ↪ tempTransactionLogFilename))
307            using (var links = new UInt64Links(memoryAdapter))
308            {
309                using (var transaction = memoryAdapter.BeginTransaction())
310                {
311                    var l1 = links.CreateAndUpdate(itself, itself);
312                    var l2 = links.CreateAndUpdate(itself, itself);
313
314                    Global.Trash = links.Update(l2, l2, l1, l2);
315
316                    links.Delete(l1);
317
318                    transaction.Commit();
319                }
320
321                Global.Trash = links.Count();
322            }
323
324            Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran↵
       ↪ sactionLogFilename);
325
326            // Damage database
327
328            FileHelpers.WriteFirst(tempTransactionLogFilename, new
       ↪ UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
329
330            // Try load damaged database
331            try
332            {
333                // TODO: Fix
334                using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
           ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
           ↪ tempTransactionLogFilename))
335                using (var links = new UInt64Links(memoryAdapter))
336                {
337                    Global.Trash = links.Count();
338                }
339            }
340            catch (NotSupportedException ex)
```

```
              {
                   Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
              ↪     yet.");
              }

              Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
              ↪     sactionLogFilename);

              File.Delete(tempDatabaseFilename);
              File.Delete(tempTransactionLogFilename);
         }

         [Fact]
         public static void Bug1Test()
         {
              var tempDatabaseFilename = Path.GetTempFileName();
              var tempTransactionLogFilename = Path.GetTempFileName();

              var itself = _constants.Itself;

              // User Code Error (Autoreverted), some data saved
              try
              {
                   ulong l1;
                   ulong l2;

                   using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                   ↪     UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                   ↪     tempTransactionLogFilename))
                   using (var links = new UInt64Links(memoryAdapter))
                   {
                        l1 = links.CreateAndUpdate(itself, itself);
                        l2 = links.CreateAndUpdate(itself, itself);

                        l2 = links.Update(l2, l2, l1, l2);

                        links.CreateAndUpdate(l2, itself);
                        links.CreateAndUpdate(l2, itself);
                   }

                   Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
                   ↪     TransactionLogFilename);

                   using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                   ↪     UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                   ↪     tempTransactionLogFilename))
                   using (var links = new UInt64Links(memoryAdapter))
                   {
                        using (var transaction = memoryAdapter.BeginTransaction())
                        {
                             l2 = links.Update(l2, l1);

                             links.Delete(l2);

                             ExceptionThrower();

                             transaction.Commit();
                        }

                        Global.Trash = links.Count();
                   }
              }
              catch
              {
                   Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
                   ↪     TransactionLogFilename);
              }

              File.Delete(tempDatabaseFilename);
              File.Delete(tempTransactionLogFilename);
         }

         private static void ExceptionThrower()
         {
              throw new Exception();
         }

         [Fact]
         public static void PathsTest()
```

```csharp
        {
            var source = _constants.SourcePart;
            var target = _constants.TargetPart;

            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                var l1 = links.CreatePoint();
                var l2 = links.CreatePoint();

                var r1 = links.GetByKeys(l1, source, target, source);
                var r2 = links.CheckPathExistance(l2, l2, l2, l2);
            }
        }

        [Fact]
        public static void RecursiveStringFormattingTest()
        {
            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.

                var a = links.CreatePoint();
                var b = links.CreatePoint();
                var c = links.CreatePoint();

                var ab = links.CreateAndUpdate(a, b);
                var cb = links.CreateAndUpdate(c, b);
                var ac = links.CreateAndUpdate(a, c);

                a = links.Update(a, c, b);
                b = links.Update(b, a, c);
                c = links.Update(c, a, b);

                Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
                Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
                Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));

                Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
                    "(5:(4:5 (6:5 4)) 6)");
                Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
                    "(6:(5:(4:5 6) 6) 4)");
                Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
                    "(4:(5:4 (6:5 4)) 6)");

                // TODO: Think how to build balanced syntax tree while formatting structure (eg.
                    "(4:(5:4 6) (6:5 4)") instead of "(4:(5:4 (6:5 4)) 6)"

                Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
                    "{{5}{5}{4}{6}}");
                Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
                    "{{5}{6}{6}{4}}");
                Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
                    "{{4}{5}{4}{6}}");
            }
        }

        private static void DefaultFormatter(StringBuilder sb, ulong link)
        {
            sb.Append(link.ToString());
        }

        #endregion

        #region Performance

        /*
        public static void RunAllPerformanceTests()
        {
            try
            {
                links.TestLinksInSteps();
            }
            catch (Exception ex)
            {
                ex.WriteToConsole();
            }
```

```
484              return;
485
486              try
487              {
488                  //ThreadPool.SetMaxThreads(2, 2);
489
490                  // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
     ↪ результат
491                  // Также это дополнительно помогает в отладке
492                  // Увеличивает вероятность попадания информации в кэши
493                  for (var i = 0; i < 10; i++)
494                  {
495                      //0 - 10 ГБ
496                      //Каждые 100 МБ срез цифр
497
498                      //links.TestGetSourceFunction();
499                      //links.TestGetSourceFunctionInParallel();
500                      //links.TestGetTargetFunction();
501                      //links.TestGetTargetFunctionInParallel();
502                      links.Create64BillionLinks();
503
504                      links.TestRandomSearchFixed();
505                      //links.Create64BillionLinksInParallel();
506                      links.TestEachFunction();
507                      //links.TestForeach();
508                      //links.TestParallelForeach();
509                  }
510
511                  links.TestDeletionOfAllLinks();
512
513              }
514              catch (Exception ex)
515              {
516                  ex.WriteToConsole();
517              }
518          }*/
519
520          /*
521      public static void TestLinksInSteps()
522      {
523              const long gibibyte = 1024 * 1024 * 1024;
524              const long mebibyte = 1024 * 1024;
525
526              var totalLinksToCreate = gibibyte /
     ↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
527              var linksStep = 102 * mebibyte /
     ↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
528
529              var creationMeasurements = new List<TimeSpan>();
530              var searchMeasuremets = new List<TimeSpan>();
531              var deletionMeasurements = new List<TimeSpan>();
532
533              GetBaseRandomLoopOverhead(linksStep);
534              GetBaseRandomLoopOverhead(linksStep);
535
536              var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
537
538              ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
539
540              var loops = totalLinksToCreate / linksStep;
541
542              for (int i = 0; i < loops; i++)
543              {
544                  creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
545                  searchMeasuremets.Add(Measure(() => links.RunRandomSearches(linksStep)));
546
547                  Console.Write("\rC + S {0}/{1}", i + 1, loops);
548              }
549
550              ConsoleHelpers.Debug();
551
552              for (int i = 0; i < loops; i++)
553              {
554                  deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
555
556                  Console.Write("\rD {0}/{1}", i + 1, loops);
557              }
558
559              ConsoleHelpers.Debug();
560
```

```csharp
                ConsoleHelpers.Debug("C S D");

                for (int i = 0; i < loops; i++)
                {
                    ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
searchMeasuremets[i], deletionMeasurements[i]);
                }

                ConsoleHelpers.Debug("C S D (no overhead)");

                for (int i = 0; i < loops; i++)
                {
                    ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] - stepLoopOverhead,
searchMeasuremets[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
                }

                ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
links.Total);
            }

        private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
amountToCreate)
        {
            for (long i = 0; i < amountToCreate; i++)
                links.Create(0, 0);
        }

         private static TimeSpan GetBaseRandomLoopOverhead(long loops)
         {
            return Measure(() =>
            {
                ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
                ulong result = 0;
                for (long i = 0; i < loops; i++)
                {
                    var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
                    var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);

                    result += maxValue + source + target;
                }
                Global.Trash = result;
            });
        }
         */

        [Fact(Skip = "performance test")]
        public static void GetSourceTest()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
                    Iterations);

                ulong counter = 0;

                //var firstLink = links.First();
                // Создаём одну связь, из которой будет производить считывание
                var firstLink = links.Create();

                var sw = Stopwatch.StartNew();

                // Тестируем саму функцию
                for (ulong i = 0; i < Iterations; i++)
                {
                    counter += links.GetSource(firstLink);
                }

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

                // Удаляем связь, из которой производилось считывание
                links.Delete(firstLink);

                ConsoleHelpers.Debug(
                    "{0} Iterations of GetSource function done in {1} ({2} Iterations per
                        second), counter result: {3}",
                    Iterations, elapsedTime, (long)iterationsPerSecond, counter);
```

```csharp
                }
            }

        [Fact(Skip = "performance test")]
        public static void GetSourceInParallel()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
                ↪   parallel.", Iterations);

                long counter = 0;

                //var firstLink = links.First();
                var firstLink = links.Create();

                var sw = Stopwatch.StartNew();

                // Тестируем саму функцию
                Parallel.For(0, Iterations, x =>
                {
                    Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
                    //Interlocked.Increment(ref counter);
                });

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

                links.Delete(firstLink);

                ConsoleHelpers.Debug(
                    "{0} Iterations of GetSource function done in {1} ({2} Iterations per
                    ↪   second), counter result: {3}",
                    Iterations, elapsedTime, (long)iterationsPerSecond, counter);
            }
        }

        [Fact(Skip = "performance test")]
        public static void TestGetTarget()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
                ↪   Iterations);

                ulong counter = 0;

                //var firstLink = links.First();
                var firstLink = links.Create();

                var sw = Stopwatch.StartNew();

                for (ulong i = 0; i < Iterations; i++)
                {
                    counter += links.GetTarget(firstLink);
                }

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

                links.Delete(firstLink);

                ConsoleHelpers.Debug(
                    "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
                    ↪   second), counter result: {3}",
                    Iterations, elapsedTime, (long)iterationsPerSecond, counter);
            }
        }

        [Fact(Skip = "performance test")]
        public static void TestGetTargetInParallel()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
```

```
709              ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
     ↪  parallel.", Iterations);

710
711              long counter = 0;
712
713              //var firstLink = links.First();
714              var firstLink = links.Create();
715
716              var sw = Stopwatch.StartNew();
717
718              Parallel.For(0, Iterations, x =>
719              {
720                  Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
721                  //Interlocked.Increment(ref counter);
722              });
723
724              var elapsedTime = sw.Elapsed;
725
726              var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
727
728              links.Delete(firstLink);
729
730              ConsoleHelpers.Debug(
731                  "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
     ↪  second), counter result: {3}",
732                  Iterations, elapsedTime, (long)iterationsPerSecond, counter);
733          }
734      }
735
736      // TODO: Заполнить базу данных перед тестом
737      /*
738      [Fact]
739      public void TestRandomSearchFixed()
740      {
741          var tempFilename = Path.GetTempFileName();
742
743          using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
     ↪  DefaultLinksSizeStep))
744          {
745              long iterations = 64 * 1024 * 1024 /
     ↪  Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
746
747              ulong counter = 0;
748              var maxLink = links.Total;
749
750              ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);
751
752              var sw = Stopwatch.StartNew();
753
754              for (var i = iterations; i > 0; i--)
755              {
756                  var source =
     ↪  RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
757                  var target =
     ↪  RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
758
759                  counter += links.Search(source, target);
760              }
761
762              var elapsedTime = sw.Elapsed;
763
764              var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
765
766              ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
     ↪  Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
     ↪  counter);
767          }
768
769          File.Delete(tempFilename);
770      }*/
771
772      [Fact(Skip = "useless: O(0), was dependent on creation tests")]
773      public static void TestRandomSearchAll()
774      {
775          using (var scope = new TempLinksTestScope())
776          {
777              var links = scope.Links;
778              ulong counter = 0;
779
780              var maxLink = links.Count();
```

```
781
782                var iterations = links.Count();
783
784                ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
        ↪   links.Count());
785
786                var sw = Stopwatch.StartNew();
787
788                for (var i = iterations; i > 0; i--)
789                {
790                    var linksAddressRange = new Range<ulong>(_constants.MinPossibleIndex,
            ↪   maxLink);
791
792                    var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
793                    var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
794
795                    counter += links.SearchOrDefault(source, target);
796                }
797
798                var elapsedTime = sw.Elapsed;
799
800                var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
801
802                ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
        ↪   Iterations per second), c: {3}",
803                    iterations, elapsedTime, (long)iterationsPerSecond, counter);
804            }
805        }
806
807        [Fact(Skip = "useless: O(0), was dependent on creation tests")]
808        public static void TestEach()
809        {
810            using (var scope = new TempLinksTestScope())
811            {
812                var links = scope.Links;
813
814                var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
815
816                ConsoleHelpers.Debug("Testing Each function.");
817
818                var sw = Stopwatch.StartNew();
819
820                links.Each(counter.IncrementAndReturnTrue);
821
822                var elapsedTime = sw.Elapsed;
823
824                var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
825
826                ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
        ↪   links per second)",
827                    counter, elapsedTime, (long)linksPerSecond);
828            }
829        }
830
831        /*
832        [Fact]
833        public static void TestForeach()
834        {
835            var tempFilename = Path.GetTempFileName();
836
837            using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪   DefaultLinksSizeStep))
838            {
839                ulong counter = 0;
840
841                ConsoleHelpers.Debug("Testing foreach through links.");
842
843                var sw = Stopwatch.StartNew();
844
845                //foreach (var link in links)
846                //{
847                //    counter++;
848                //}
849
850                var elapsedTime = sw.Elapsed;
851
852                var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
853
854                ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
    ↪   links per second)", counter, elapsedTime, (long)linksPerSecond);
```

```
855                 }

856
857             File.Delete(tempFilename);
858         }
859         */

860
861         /*
862         [Fact]
863         public static void TestParallelForeach()
864         {
865             var tempFilename = Path.GetTempFileName();

866
867             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪   DefaultLinksSizeStep))
868             {

869
870                 long counter = 0;

871
872                 ConsoleHelpers.Debug("Testing parallel foreach through links.");

873
874                 var sw = Stopwatch.StartNew();

875
876                 //Parallel.ForEach((IEnumerable<ulong>)links, x =>
877                 //{
878                 //    Interlocked.Increment(ref counter);
879                 //});

880
881                 var elapsedTime = sw.Elapsed;

882
883                 var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;

884
885                 ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
    ↪   {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
886             }

887
888             File.Delete(tempFilename);
889         }
890         */

891
892         [Fact(Skip = "performance test")]
893         public static void Create64BillionLinks()
894         {
895             using (var scope = new TempLinksTestScope())
896             {
897                 var links = scope.Links;
898                 var linksBeforeTest = links.Count();

899
900                 long linksToCreate = 64 * 1024 * 1024 /
    ↪   UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;

901
902                 ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);

903
904                 var elapsedTime = Performance.Measure(() =>
905                 {
906                     for (long i = 0; i < linksToCreate; i++)
907                     {
908                         links.Create();
909                     }
910                 });

911
912                 var linksCreated = links.Count() - linksBeforeTest;
913                 var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;

914
915                 ConsoleHelpers.Debug("Current links count: {0}.", links.Count());

916
917                 ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
    ↪   linksCreated, elapsedTime,
918                     (long)linksPerSecond);
919             }
920         }

921
922         [Fact(Skip = "performance test")]
923         public static void Create64BillionLinksInParallel()
924         {
925             using (var scope = new TempLinksTestScope())
926             {
927                 var links = scope.Links;
928                 var linksBeforeTest = links.Count();

929
```

```
930              var sw = Stopwatch.StartNew();

932              long linksToCreate = 64 * 1024 * 1024 /
                 ↪   UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;

934              ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);

936              Parallel.For(0, linksToCreate, x => links.Create());

938              var elapsedTime = sw.Elapsed;

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

943              ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
                 ↪   linksCreated, elapsedTime,
944                  (long)linksPerSecond);
945          }
946      }

948      [Fact(Skip = "useless: O(0), was dependent on creation tests")]
949      public static void TestDeletionOfAllLinks()
950      {
951          using (var scope = new TempLinksTestScope())
952          {
953              var links = scope.Links;
954              var linksBeforeTest = links.Count();

956              ConsoleHelpers.Debug("Deleting all links");

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

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

963              ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
                 ↪   linksDeleted, elapsedTime,
964                  (long)linksPerSecond);
965          }
966      }

968      #endregion
969  }
970  }
```

./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;

16  namespace Platform.Data.Doublets.Tests
17  {
18      public static class OptimalVariantSequenceTests
19      {
20          private const string SequenceExample = "зеленела зелёная зелень";

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;

30                  links.UseUnicode();

32                  var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);

34                  var meaningRoot = links.CreatePoint();
35                  var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
```

```csharp
                        var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
                        var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
                        ↪    constants.Itself);

                        var unaryNumberToAddressConverter = new
                        ↪    UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
                        var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
                        var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                        ↪    frequencyMarker, unaryOne, unaryNumberIncrementer);
                        var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
                        ↪    frequencyPropertyMarker, frequencyMarker);
                        var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
                        ↪    frequencyPropertyOperator, frequencyIncrementer);
                        var linkToItsFrequencyNumberConverter = new
                        ↪    LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                        ↪    unaryNumberToAddressConverter);
                        var sequenceToItsLocalElementLevelsConverter = new
                        ↪    SequenceToItsLocalElementLevelsConverter<ulong>(links,
                        ↪    linkToItsFrequencyNumberConverter);
                        var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                        ↪    sequenceToItsLocalElementLevelsConverter);

                        var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
                        ↪    Walker = new LeveledSequenceWalker<ulong>(links) });

                        ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
                        ↪    index, optimalVariantConverter);
                }
        }

        [Fact]
        public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
        {
                using (var scope = new TempLinksTestScope(useSequences: false))
                {
                        var links = scope.Links;

                        links.UseUnicode();

                        var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);

                        var linksToFrequencies = new Dictionary<ulong, ulong>();

                        var totalSequenceSymbolFrequencyCounter = new
                        ↪    TotalSequenceSymbolFrequencyCounter<ulong>(links);

                        var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
                        ↪    totalSequenceSymbolFrequencyCounter);

                        var index = new
                        ↪    CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
                        var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque⌋
                        ↪    ncyNumberConverter<ulong>(linkFrequenciesCache);

                        var sequenceToItsLocalElementLevelsConverter = new
                        ↪    SequenceToItsLocalElementLevelsConverter<ulong>(links,
                        ↪    linkToItsFrequencyNumberConverter);
                        var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                        ↪    sequenceToItsLocalElementLevelsConverter);

                        var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
                        ↪    Walker = new LeveledSequenceWalker<ulong>(links) });

                        ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
                        ↪    index, optimalVariantConverter);
                }
        }

        private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
        ↪    SequenceToItsLocalElementLevelsConverter<ulong>
        ↪    sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
        ↪    OptimalVariantConverter<ulong> optimalVariantConverter)
        {
                index.Add(sequence);

                var optimalVariant = optimalVariantConverter.Convert(sequence);

                var readSequence1 = sequences.ToList(optimalVariant);
```

```
90
91                    Assert.True(sequence.SequenceEqual(readSequence1));
92                }
93            }
94    }
```

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

```csharp
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>() {
                     ↪  Walker = new LeveledSequenceWalker<ulong>(links) });;;
24
25                   var sequence = new ulong[sequenceLength];
26                   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                   sw3.Stop();
47
48                   Assert.True(sequence.SequenceEqual(readSequence1));
49
50                   Assert.True(sequence.SequenceEqual(readSequence2));
51
52                   // Assert.True(sw2.Elapsed < sw3.Elapsed);
53
54                   Console.WriteLine($"Stack-based walker: {sw3.Elapsed}, Level-based reader:
                     ↪  {sw2.Elapsed}");
55
56                   for (var i = 0; i < sequenceLength; i++)
57                   {
58                       links.Delete(sequence[i]);
59                   }
60               }
61           }
62       }
63   }
```

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

```csharp
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;
```

```csharp
namespace Platform.Data.Doublets.Tests
{
    public static class ResizableDirectMemoryLinksTests
    {
        private static readonly LinksCombinedConstants<ulong, ulong, int> _constants =
          Default<LinksCombinedConstants<ulong, ulong, int>>.Instance;

        [Fact]
        public static void BasicFileMappedMemoryTest()
        {
            var tempFilename = Path.GetTempFileName();
            using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(tempFilename))
            {
                memoryAdapter.TestBasicMemoryOperations();
            }
            File.Delete(tempFilename);
        }

        [Fact]
        public static void BasicHeapMemoryTest()
        {
            using (var memory = new
              HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
            using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
              UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
            {
                memoryAdapter.TestBasicMemoryOperations();
            }
        }

        private static void TestBasicMemoryOperations(this ILinks<ulong> memoryAdapter)
        {
            var link = memoryAdapter.Create();
            memoryAdapter.Delete(link);
        }

        [Fact]
        public static void NonexistentReferencesHeapMemoryTest()
        {
            using (var memory = new
              HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
            using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
              UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
            {
                memoryAdapter.TestNonexistentReferences();
            }
        }

        private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
        {
            var link = memoryAdapter.Create();
            memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
            var resultLink = _constants.Null;
            memoryAdapter.Each(foundLink =>
            {
                resultLink = foundLink[_constants.IndexPart];
                return _constants.Break;
            }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
            Assert.True(resultLink == link);
            Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
            memoryAdapter.Delete(link);
        }
    }
}
```

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

```csharp
using Xunit;
using Platform.Scopes;
using Platform.Memory;
using Platform.Data.Doublets.ResizableDirectMemory;
using Platform.Data.Doublets.Decorators;

namespace Platform.Data.Doublets.Tests
{
    public static class ScopeTests
    {
        [Fact]
        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 =
               ↪  Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
22
23          static SequencesTests()
24          {
25              // Trigger static constructor to not mess with perfomance measurements
26              _ = BitString.GetBitMaskFromIndex(1);
27          }
28
29          [Fact]
30          public static void CreateAllVariantsTest()
31          {
32              const long sequenceLength = 8;
33
34              using (var scope = new TempLinksTestScope(useSequences: true))
35              {
36                  var links = scope.Links;
37                  var sequences = scope.Sequences;
38
39                  var sequence = new ulong[sequenceLength];
40                  for (var i = 0; i < sequenceLength; i++)
41                  {
42                      sequence[i] = links.Create();
43                  }
44
45                  var sw1 = Stopwatch.StartNew();
```

```csharp
            var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();

            var sw2 = Stopwatch.StartNew();
            var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();

            Assert.True(results1.Count > results2.Length);
            Assert.True(sw1.Elapsed > sw2.Elapsed);

            for (var i = 0; i < sequenceLength; i++)
            {
                links.Delete(sequence[i]);
            }

            Assert.True(links.Count() == 0);
        }
    }

    //[Fact]
    //public void CUDTest()
    //{
    //    var tempFilename = Path.GetTempFileName();

    //    const long sequenceLength = 8;

    //    const ulong itself = LinksConstants.Itself;

    //    using (var memoryAdapter = new ResizableDirectMemoryLinks(tempFilename,
    //  DefaultLinksSizeStep))
    //    using (var links = new Links(memoryAdapter))
    //    {
    //        var sequence = new ulong[sequenceLength];
    //        for (var i = 0; i < sequenceLength; i++)
    //            sequence[i] = links.Create(itself, itself);


    //        SequencesOptions o = new SequencesOptions();

    // TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
    //            o.


    //        var sequences = new Sequences(links);

    //        var sw1 = Stopwatch.StartNew();
    //        var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();

    //        var sw2 = Stopwatch.StartNew();
    //        var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();

    //        Assert.True(results1.Count > results2.Length);
    //        Assert.True(sw1.Elapsed > sw2.Elapsed);

    //        for (var i = 0; i < sequenceLength; i++)
    //            links.Delete(sequence[i]);
    //    }

    //    File.Delete(tempFilename);
    //}

    [Fact]
    public static void AllVariantsSearchTest()
    {
        const long sequenceLength = 8;

        using (var scope = new TempLinksTestScope(useSequences: true))
        {
            var links = scope.Links;
            var sequences = scope.Sequences;

            var sequence = new ulong[sequenceLength];
            for (var i = 0; i < sequenceLength; i++)
            {
                sequence[i] = links.Create();
            }

            var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();

            //for (int i = 0; i < createResults.Length; i++)
            //    sequences.Create(createResults[i]);
```

```csharp
                    var sw0 = Stopwatch.StartNew();
                    var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();

                    var sw1 = Stopwatch.StartNew();
                    var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();

                    var sw2 = Stopwatch.StartNew();
                    var searchResults2 = sequences.Each1(sequence); sw2.Stop();

                    var sw3 = Stopwatch.StartNew();
                    var searchResults3 = sequences.Each(sequence); sw3.Stop();

                    var intersection0 = createResults.Intersect(searchResults0).ToList();
                    Assert.True(intersection0.Count == searchResults0.Count);
                    Assert.True(intersection0.Count == createResults.Length);

                    var intersection1 = createResults.Intersect(searchResults1).ToList();
                    Assert.True(intersection1.Count == searchResults1.Count);
                    Assert.True(intersection1.Count == createResults.Length);

                    var intersection2 = createResults.Intersect(searchResults2).ToList();
                    Assert.True(intersection2.Count == searchResults2.Count);
                    Assert.True(intersection2.Count == createResults.Length);

                    var intersection3 = createResults.Intersect(searchResults3).ToList();
                    Assert.True(intersection3.Count == searchResults3.Count);
                    Assert.True(intersection3.Count == createResults.Length);

                    for (var i = 0; i < sequenceLength; i++)
                    {
                        links.Delete(sequence[i]);
                    }
                }
            }

            [Fact]
            public static void BalancedVariantSearchTest()
            {
                const long sequenceLength = 200;

                using (var scope = new TempLinksTestScope(useSequences: true))
                {
                    var links = scope.Links;
                    var sequences = scope.Sequences;

                    var sequence = new ulong[sequenceLength];
                    for (var i = 0; i < sequenceLength; i++)
                    {
                        sequence[i] = links.Create();
                    }

                    var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);

                    var sw1 = Stopwatch.StartNew();
                    var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();

                    var sw2 = Stopwatch.StartNew();
                    var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();

                    var sw3 = Stopwatch.StartNew();
                    var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();

                    // На количестве в 200 элементов это будет занимать вечность
                    //var sw4 = Stopwatch.StartNew();
                    //var searchResults4 = sequences.Each(sequence); sw4.Stop();

                    Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);

                    Assert.True(searchResults3.Count == 1 && balancedVariant ==
                    ↪  searchResults3.First());

                    //Assert.True(sw1.Elapsed < sw2.Elapsed);

                    for (var i = 0; i < sequenceLength; i++)
                    {
                        links.Delete(sequence[i]);
                    }
                }
            }
```

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

```csharp
                    var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);

                    var balancedVariant = balancedVariantConverter.Convert(sequence);

                    var partialSequence = new ulong[sequenceLength - 2];

                    Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);

                    var sw1 = Stopwatch.StartNew();
                    var searchResults1 =
                    ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();

                    var sw2 = Stopwatch.StartNew();
                    var searchResults2 =
                    ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();

                    Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);

                    Assert.True(searchResults2.Count == 1 && balancedVariant ==
                    ↪ searchResults2.First());

                    for (var i = 0; i < sequenceLength; i++)
                    {
                        links.Delete(sequence[i]);
                    }
                }
            }

            [Fact(Skip = "Correct implementation is pending")]
            public static void PatternMatchTest()
            {
                var zeroOrMany = Sequences.Sequences.ZeroOrMany;

                using (var scope = new TempLinksTestScope(useSequences: true))
                {
                    var links = scope.Links;
                    var sequences = scope.Sequences;

                    var e1 = links.Create();
                    var e2 = links.Create();

                    var sequence = new[]
                    {
                        e1, e2, e1, e2 // mama / papa
                    };

                    var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);

                    var balancedVariant = balancedVariantConverter.Convert(sequence);

                    // 1: [1]
                    // 2: [2]
                    // 3: [1,2]
                    // 4: [1,2,1,2]

                    var doublet = links.GetSource(balancedVariant);

                    var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);

                    Assert.True(matchedSequences1.Count == 0);

                    var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);

                    Assert.True(matchedSequences2.Count == 0);

                    var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);

                    Assert.True(matchedSequences3.Count == 0);

                    var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);

                    Assert.Contains(doublet, matchedSequences4);
                    Assert.Contains(balancedVariant, matchedSequences4);

                    for (var i = 0; i < sequence.Length; i++)
                    {
                        links.Delete(sequence[i]);
                    }
                }
            }
```

```csharp
355
356            [Fact]
357            public static void IndexTest()
358            {
359                using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
                        true }, useSequences: true))
360                {
361                    var links = scope.Links;
362                    var sequences = scope.Sequences;
363                    var index = sequences.Options.Index;
364
365                    var e1 = links.Create();
366                    var e2 = links.Create();
367
368                    var sequence = new[]
369                    {
370                        e1, e2, e1, e2 // mama / papa
371                    };
372
373                    Assert.False(index.MightContain(sequence));
374
375                    index.Add(sequence);
376
377                    Assert.True(index.MightContain(sequence));
378                }
379            }
380
381        /// <summary>Imported from https://raw.githubusercontent.com/wiki/Konard/LinksPlatform/%
                D0%9E-%D1%82%D0%BE%D0%BC%2C-%D0%BA%D0%B0%D0%BA-%D0%B2%D1%81%D1%91-%D0%BD%D0%B0%D1%87
                %D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>
382        private static readonly string _exampleText =
383            @"([english
                version](https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))
384
385 Обозначение пустоты, какое оно? Темнота ли это? Там где отсутствие света, отсутствие фотонов
        (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?
386
387 [![чёрное пространство, белое
        пространство](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png
        ""чёрное пространство, белое пространство"")](https://raw.githubusercontent.com/Konard/Links
        Platform/master/doc/Intro/1.png)
388
389 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?
390
391 [![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)
392
393 А что если повторить? Сделать копию? Создать дубликат? Из одного сделать два? Может это быть
        так? Инверсия? Отражение? Сумма?
394
395 [![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)
396
397 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?
398
399 [![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)
400
401 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?
402
403 [![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)
```

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?

[![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)

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?

[![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)

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?

[![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)

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?

[![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)

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?

[![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)

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

[![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)

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

[![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)

...

[![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animation-500.gif ""анимация"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animation-500.gif)";

```csharp
        private static readonly string _exampleLoremIpsumText =
            @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
              incididunt ut labore et dolore magna aliqua.
Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
   consequat.";

        [Fact]
        public static void CompressionTest()
        {
            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;

                var e1 = links.Create();
                var e2 = links.Create();

                var sequence = new[]
                {
                    e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
                };

                var balancedVariantConverter = new BalancedVariantConverter<ulong>(links.Unsync);
                var totalSequenceSymbolFrequencyCounter = new
                    TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
                var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync,
                    totalSequenceSymbolFrequencyCounter);
                var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
                    balancedVariantConverter, doubletFrequenciesCache);

                var compressedVariant = compressingConverter.Convert(sequence);

                // 1: [1]         (1->1) point
                // 2: [2]         (2->2) point
                // 3: [1,2]       (1->2) doublet
                // 4: [1,2,1,2] (3->3) doublet

                Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
                Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
                Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
                Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);

                var source = _constants.SourcePart;
                var target = _constants.TargetPart;

                Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
                Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
                Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
                Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);

                // 4 - length of sequence
                Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0)
                    == sequence[0]);
                Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
                    == sequence[1]);
                Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
                    == sequence[2]);
                Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
                    == sequence[3]);
            }
        }

        [Fact]
        public static void CompressionEfficiencyTest()
        {
            var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
                StringSplitOptions.RemoveEmptyEntries);
            var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
            var totalCharacters = arrays.Select(x => x.Length).Sum();

            using (var scope1 = new TempLinksTestScope(useSequences: true))
            using (var scope2 = new TempLinksTestScope(useSequences: true))
            using (var scope3 = new TempLinksTestScope(useSequences: true))
            {
                scope1.Links.Unsync.UseUnicode();
                scope2.Links.Unsync.UseUnicode();
                scope3.Links.Unsync.UseUnicode();
```

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

```csharp
                {
                    compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
                }

                var elapsed2 = sw2.Elapsed;

                for (int i = START; i < END; i++)
                {
                    linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
                }

                var initialCount3 = scope3.Links.Unsync.Count();

                var sw3 = Stopwatch.StartNew();

                for (int i = START; i < END; i++)
                {
                    //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
                    compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
                }

                var elapsed3 = sw3.Elapsed;

                Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
                ↪   Optimal variant: {elapsed3}");

                // Assert.True(elapsed1 > elapsed2);

                // Checks
                for (int i = START; i < END; i++)
                {
                    var sequence1 = compressed1[i];
                    var sequence2 = compressed2[i];
                    var sequence3 = compressed3[i];

                    var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                    ↪   scope1.Links.Unsync);

                    var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                    ↪   scope2.Links.Unsync);

                    var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
                    ↪   scope3.Links.Unsync);

                    var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
                    ↪   link.IsPartialPoint());
                    var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
                    ↪   link.IsPartialPoint());
                    var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
                    ↪   link.IsPartialPoint());

                    //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
                    ↪   arrays[i].Length > 3)
                    //    Assert.False(structure1 == structure2);
                    //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
                    ↪   arrays[i].Length > 3)
                    //    Assert.False(structure3 == structure2);

                    Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
                    Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
                }

                Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
                ↪   totalCharacters);
                Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
                ↪   totalCharacters);
                Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
                ↪   totalCharacters);

                Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
                ↪   totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) /
                ↪   totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) /
                ↪   totalCharacters}");

                Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
                ↪   scope2.Links.Unsync.Count() - initialCount2);
                Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
                ↪   scope2.Links.Unsync.Count() - initialCount2);
```

```csharp
                    var duplicateProvider1 = new
                    ↪   DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
                    var duplicateProvider2 = new
                    ↪   DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
                    var duplicateProvider3 = new
                    ↪   DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);

                    var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
                    var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
                    var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);

                    var duplicates1 = duplicateCounter1.Count();

                    ConsoleHelpers.Debug("------");

                    var duplicates2 = duplicateCounter2.Count();

                    ConsoleHelpers.Debug("------");

                    var duplicates3 = duplicateCounter3.Count();

                    Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");

                    linkFrequenciesCache1.ValidateFrequencies();
                    linkFrequenciesCache3.ValidateFrequencies();
                }
            }

            [Fact]
            public static void CompressionStabilityTest()
            {
                // TODO: Fix bug (do a separate test)
                //const ulong minNumbers = 0;
                //const ulong maxNumbers = 1000;

                const ulong minNumbers = 10000;
                const ulong maxNumbers = 12500;

                var strings = new List<string>();

                for (ulong i = minNumbers; i < maxNumbers; i++)
                {
                    strings.Add(i.ToString());
                }

                var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
                var totalCharacters = arrays.Select(x => x.Length).Sum();

                using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
                ↪   SequencesOptions<ulong> { UseCompression = true,
                ↪   EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
                using (var scope2 = new TempLinksTestScope(useSequences: true))
                {
                    scope1.Links.UseUnicode();
                    scope2.Links.UseUnicode();

                    //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
                    var compressor1 = scope1.Sequences;
                    var compressor2 = scope2.Sequences;

                    var compressed1 = new ulong[arrays.Length];
                    var compressed2 = new ulong[arrays.Length];

                    var sw1 = Stopwatch.StartNew();

                    var START = 0;
                    var END = arrays.Length;

                    // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
                    // Stability issue starts at 10001 or 11000
                    //for (int i = START; i < END; i++)
                    //{
                    //    var first = compressor1.Compress(arrays[i]);
                    //    var second = compressor1.Compress(arrays[i]);

                    //    if (first == second)
                    //        compressed1[i] = first;
                    //    else
                    //    {
```

```csharp
701    //        // TODO: Find a solution for this case
702    //    }
703    //}

705    for (int i = START; i < END; i++)
706    {
707        var first = compressor1.Create(arrays[i]);
708        var second = compressor1.Create(arrays[i]);

710        if (first == second)
711        {
712            compressed1[i] = first;
713        }
714        else
715        {
716            // TODO: Find a solution for this case
717        }
718    }

720    var elapsed1 = sw1.Elapsed;

722    var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);

724    var sw2 = Stopwatch.StartNew();

726    for (int i = START; i < END; i++)
727    {
728        var first = balancedVariantConverter.Convert(arrays[i]);
729        var second = balancedVariantConverter.Convert(arrays[i]);

731        if (first == second)
732        {
733            compressed2[i] = first;
734        }
735    }

737    var elapsed2 = sw2.Elapsed;

739    Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
       ↪  {elapsed2}");

741    Assert.True(elapsed1 > elapsed2);

743    // Checks
744    for (int i = START; i < END; i++)
745    {
746        var sequence1 = compressed1[i];
747        var sequence2 = compressed2[i];

749        if (sequence1 != _constants.Null && sequence2 != _constants.Null)
750        {
751            var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
               ↪  scope1.Links);

753            var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
               ↪  scope2.Links);

755            //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
               ↪  link.IsPartialPoint());
756            //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
               ↪  link.IsPartialPoint());

758            //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
               ↪  arrays[i].Length > 3)
759            //    Assert.False(structure1 == structure2);

761            Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
762        }
763    }

765    Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
766    Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);

768    Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
       ↪  totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
       ↪  totalCharacters}");

770    Assert.True(scope1.Links.Count() <= scope2.Links.Count());
771
```

```csharp
                //compressor1.ValidateFrequencies();
            }
        }

        [Fact]
        public static void RandomNumbersCompressionQualityTest()
        {
            const ulong N = 500;

            //const ulong minNumbers = 10000;
            //const ulong maxNumbers = 20000;

            //var strings = new List<string>();

            //for (ulong i = 0; i < N; i++)
            //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
            ↪   maxNumbers).ToString());

            var strings = new List<string>();

            for (ulong i = 0; i < N; i++)
            {
                strings.Add(RandomHelpers.Default.NextUInt64().ToString());
            }

            strings = strings.Distinct().ToList();

            var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
            var totalCharacters = arrays.Select(x => x.Length).Sum();

            using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
            ↪   SequencesOptions<ulong> { UseCompression = true,
            ↪   EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
            using (var scope2 = new TempLinksTestScope(useSequences: true))
            {
                scope1.Links.UseUnicode();
                scope2.Links.UseUnicode();

                var compressor1 = scope1.Sequences;
                var compressor2 = scope2.Sequences;

                var compressed1 = new ulong[arrays.Length];
                var compressed2 = new ulong[arrays.Length];

                var sw1 = Stopwatch.StartNew();

                var START = 0;
                var END = arrays.Length;

                for (int i = START; i < END; i++)
                {
                    compressed1[i] = compressor1.Create(arrays[i]);
                }

                var elapsed1 = sw1.Elapsed;

                var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);

                var sw2 = Stopwatch.StartNew();

                for (int i = START; i < END; i++)
                {
                    compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
                }

                var elapsed2 = sw2.Elapsed;

                Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
                ↪   {elapsed2}");

                Assert.True(elapsed1 > elapsed2);

                // Checks
                for (int i = START; i < END; i++)
                {
                    var sequence1 = compressed1[i];
                    var sequence2 = compressed2[i];

                    if (sequence1 != _constants.Null && sequence2 != _constants.Null)
                    {
```

```
848              var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                 ↪   scope1.Links);
849
850              var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                 ↪   scope2.Links);
851
852              Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
853          }
854      }
855
856      Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
857      Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
858
859      Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
         ↪   totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
         ↪   totalCharacters}");
860
861      // Can be worse than balanced variant
862      //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
863
864      //compressor1.ValidateFrequencies();
865      }
866  }
867
868  [Fact]
869  public static void AllTreeBreakDownAtSequencesCreationBugTest()
870  {
871      // Made out of AllPossibleConnectionsTest test.
872
873      //const long sequenceLength = 5; //100% bug
874      const long sequenceLength = 4; //100% bug
875      //const long sequenceLength = 3; //100% _no_bug (ok)
876
877      using (var scope = new TempLinksTestScope(useSequences: true))
878      {
879          var links = scope.Links;
880          var sequences = scope.Sequences;
881
882          var sequence = new ulong[sequenceLength];
883          for (var i = 0; i < sequenceLength; i++)
884          {
885              sequence[i] = links.Create();
886          }
887
888          var createResults = sequences.CreateAllVariants2(sequence);
889
890          Global.Trash = createResults;
891
892          for (var i = 0; i < sequenceLength; i++)
893          {
894              links.Delete(sequence[i]);
895          }
896      }
897  }
898
899  [Fact]
900  public static void AllPossibleConnectionsTest()
901  {
902      const long sequenceLength = 5;
903
904      using (var scope = new TempLinksTestScope(useSequences: true))
905      {
906          var links = scope.Links;
907          var sequences = scope.Sequences;
908
909          var sequence = new ulong[sequenceLength];
910          for (var i = 0; i < sequenceLength; i++)
911          {
912              sequence[i] = links.Create();
913          }
914
915          var createResults = sequences.CreateAllVariants2(sequence);
916          var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
917
918          for (var i = 0; i < 1; i++)
919          {
920              var sw1 = Stopwatch.StartNew();
921              var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
922
```

```
923                         var sw2 = Stopwatch.StartNew();
924                         var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();
925
926                         var sw3 = Stopwatch.StartNew();
927                         var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();
928
929                         var sw4 = Stopwatch.StartNew();
930                         var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();
931
932                         Global.Trash = searchResults3;
933                         Global.Trash = searchResults4; //-V3008
934
935                         var intersection1 = createResults.Intersect(searchResults1).ToList();
936                         Assert.True(intersection1.Count == createResults.Length);
937
938                         var intersection2 = reverseResults.Intersect(searchResults1).ToList();
939                         Assert.True(intersection2.Count == reverseResults.Length);
940
941                         var intersection0 = searchResults1.Intersect(searchResults2).ToList();
942                         Assert.True(intersection0.Count == searchResults2.Count);
943
944                         var intersection3 = searchResults2.Intersect(searchResults3).ToList();
945                         Assert.True(intersection3.Count == searchResults3.Count);
946
947                         var intersection4 = searchResults3.Intersect(searchResults4).ToList();
948                         Assert.True(intersection4.Count == searchResults4.Count);
949                     }
950
951                     for (var i = 0; i < sequenceLength; i++)
952                     {
953                         links.Delete(sequence[i]);
954                     }
955                 }
956             }
957
958             [Fact(Skip = "Correct implementation is pending")]
959             public static void CalculateAllUsagesTest()
960             {
961                 const long sequenceLength = 3;
962
963                 using (var scope = new TempLinksTestScope(useSequences: true))
964                 {
965                     var links = scope.Links;
966                     var sequences = scope.Sequences;
967
968                     var sequence = new ulong[sequenceLength];
969                     for (var i = 0; i < sequenceLength; i++)
970                     {
971                         sequence[i] = links.Create();
972                     }
973
974                     var createResults = sequences.CreateAllVariants2(sequence);
975
976                     //var reverseResults =
977                     ↪    sequences.CreateAllVariants2(sequence.Reverse().ToArray());
978                     for (var i = 0; i < 1; i++)
979                     {
980                         var linksTotalUsages1 = new ulong[links.Count() + 1];
981
982                         sequences.CalculateAllUsages(linksTotalUsages1);
983
984                         var linksTotalUsages2 = new ulong[links.Count() + 1];
985
986                         sequences.CalculateAllUsages2(linksTotalUsages2);
987
988                         var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
989                         Assert.True(intersection1.Count == linksTotalUsages2.Length);
990                     }
991
992                     for (var i = 0; i < sequenceLength; i++)
993                     {
994                         links.Delete(sequence[i]);
995                     }
996                 }
997             }
998         }
999     }
```

```csharp
using System.IO;
using Platform.Disposables;
using Platform.Data.Doublets.ResizableDirectMemory;
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Decorators;

namespace Platform.Data.Doublets.Tests
{
    public class TempLinksTestScope : DisposableBase
    {
        public readonly ILinks<ulong> MemoryAdapter;
        public readonly SynchronizedLinks<ulong> Links;
        public readonly Sequences.Sequences Sequences;
        public readonly string TempFilename;
        public readonly string TempTransactionLogFilename;
        private readonly bool _deleteFiles;

        public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool useLog = false)
            : this(new SequencesOptions<ulong>(), deleteFiles, useSequences, useLog)
        {
        }

        public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles = true, bool useSequences = false, bool useLog = false)
        {
            _deleteFiles = deleteFiles;
            TempFilename = Path.GetTempFileName();
            TempTransactionLogFilename = Path.GetTempFileName();

            var coreMemoryAdapter = new UInt64ResizableDirectMemoryLinks(TempFilename);

            MemoryAdapter = useLog ? (ILinks<ulong>)new
                UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
                coreMemoryAdapter;

            Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
            if (useSequences)
            {
                Sequences = new Sequences.Sequences(Links, sequencesOptions);
            }
        }

        protected override void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                Links.Unsync.DisposeIfPossible();
                if (_deleteFiles)
                {
                    DeleteFiles();
                }
            }
        }

        public void DeleteFiles()
        {
            File.Delete(TempFilename);
            File.Delete(TempTransactionLogFilename);
        }
    }
}
```

```csharp
using Xunit;
using Platform.Random;
using Platform.Data.Doublets.Numbers.Unary;

namespace Platform.Data.Doublets.Tests
{
    public static class UnaryNumberConvertersTests
    {
        [Fact]
        public static void ConvertersTest()
        {
            using (var scope = new TempLinksTestScope())
            {
                const int N = 10;
                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 Platform.Data.Doublets.Incrementers;
2  using Platform.Data.Doublets.Numbers.Unary;
3  using Platform.Data.Doublets.PropertyOperators;
4  using Platform.Data.Doublets.Sequences.Converters;
5  using Platform.Data.Doublets.Sequences.Indexes;
6  using Platform.Data.Doublets.Sequences.Walkers;
7  using Platform.Data.Doublets.Unicode;
8  using Xunit;
9
10 namespace Platform.Data.Doublets.Tests
11 {
12     public static class UnicodeConvertersTests
13     {
14         [Fact]
15         public static void CharAndUnicodeSymbolConvertersTest()
16         {
17             using (var scope = new TempLinksTestScope())
18             {
19                 var links = scope.Links;
20
21                 var itself = links.Constants.Itself;
22
23                 var meaningRoot = links.CreatePoint();
24                 var one = links.CreateAndUpdate(meaningRoot, itself);
25                 var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
26
27                 var powerOf2ToUnaryNumberConverter = new
   ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, one);
28                 var addressToUnaryNumberConverter = new
   ↪  AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
29                 var charToUnicodeSymbolConverter = new
   ↪  CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
   ↪  unicodeSymbolMarker);
30
31                 var originalCharacter = 'H';
32
33                 var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);
34
35                 var unaryNumberToAddressConverter = new
   ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links,
   ↪  powerOf2ToUnaryNumberConverter);
36                 var unicodeSymbolCriterionMatcher = new
   ↪  UnicodeSymbolCriterionMatcher<ulong>(links, unicodeSymbolMarker);
37                 var unicodeSymbolToCharConverter = new
   ↪  UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
   ↪  unicodeSymbolCriterionMatcher);
38
```

```csharp
39                    var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
40
41                    Assert.Equal(originalCharacter, resultingCharacter);
42                }
43            }
44
45            [Fact]
46            public static void StringAndUnicodeSequenceConvertersTest()
47            {
48                using (var scope = new TempLinksTestScope())
49                {
50                    var links = scope.Links;
51
52                    var itself = links.Constants.Itself;
53
54                    var meaningRoot = links.CreatePoint();
55                    var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
56                    var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
57                    var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
58                    var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
59                    var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);
60
61                    var powerOf2ToUnaryNumberConverter = new
                    ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
62                    var addressToUnaryNumberConverter = new
                    ↪ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
63                    var charToUnicodeSymbolConverter = new
                    ↪ CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
                    ↪ unicodeSymbolMarker);
64
65                    var unaryNumberToAddressConverter = new
                    ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links,
                    ↪ powerOf2ToUnaryNumberConverter);
66                    var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
67                    var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                    ↪ frequencyMarker, unaryOne, unaryNumberIncrementer);
68                    var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
                    ↪ frequencyPropertyMarker, frequencyMarker);
69                    var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
                    ↪ frequencyPropertyOperator, frequencyIncrementer);
70                    var linkToItsFrequencyNumberConverter = new
                    ↪ LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                    ↪ unaryNumberToAddressConverter);
71                    var sequenceToItsLocalElementLevelsConverter = new
                    ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
                    ↪ linkToItsFrequencyNumberConverter);
72                    var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                    ↪ sequenceToItsLocalElementLevelsConverter);
73
74                    var stringToUnicodeSymbolConverter = new
                    ↪ StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
                    ↪ index, optimalVariantConverter, unicodeSequenceMarker);
75
76                    var originalString = "Hello";
77
78                    var unicodeSequenceLink = stringToUnicodeSymbolConverter.Convert(originalString);
79
80                    var unicodeSymbolCriterionMatcher = new
                    ↪ UnicodeSymbolCriterionMatcher<ulong>(links, unicodeSymbolMarker);
81                    var unicodeSymbolToCharConverter = new
                    ↪ UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
                    ↪ unicodeSymbolCriterionMatcher);
82
83                    var unicodeSequenceCriterionMatcher = new
                    ↪ UnicodeSequenceCriterionMatcher<ulong>(links, unicodeSequenceMarker);
84
85                    var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
                    ↪ unicodeSymbolCriterionMatcher.IsMatched);
86
87                    var unicodeSequenceToStringConverter = new
                    ↪ UnicodeSequenceToStringConverter<ulong>(links,
                    ↪ unicodeSequenceCriterionMatcher, sequenceWalker,
                    ↪ unicodeSymbolToCharConverter);
88
89                    var resultingString =
                    ↪ unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
90
91                    Assert.Equal(originalString, resultingString);
```

```
92                }
93              }
94          }
95      }
```

Index