LinksPlatform's Platform.Data.Doublets Class Library

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

```csharp
namespace Platform.Data.Doublets.Decorators
{
    public class LinksCascadeUniquenessAndUsagesResolver<TLink> : LinksUniquenessResolver<TLink>
    {
        public LinksCascadeUniquenessAndUsagesResolver(ILinks<TLink> links) : base(links) { }

        protected override TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
        ↪  newLinkAddress)
        {
            Links.MergeUsages(oldLinkAddress, newLinkAddress);
            return base.ResolveAddressChangeConflict(oldLinkAddress, newLinkAddress);
        }
    }
}
```

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

```csharp
namespace Platform.Data.Doublets.Decorators
{
    /// <remarks>
    /// <para>Must be used in conjunction with NonNullContentsLinkDeletionResolver.</para>
    /// <para>Должен использоваться вместе с NonNullContentsLinkDeletionResolver.</para>
    /// </remarks>
    public class LinksCascadeUsagesResolver<TLink> : LinksDecoratorBase<TLink>
    {
        public LinksCascadeUsagesResolver(ILinks<TLink> links) : base(links) { }

        public override void Delete(TLink linkIndex)
        {
            this.DeleteAllUsages(linkIndex);
            Links.Delete(linkIndex);
        }
    }
}
```

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

```csharp
using System;
using System.Collections.Generic;
using Platform.Data.Constants;

namespace Platform.Data.Doublets.Decorators
{
    public abstract class LinksDecoratorBase<TLink> : LinksOperatorBase<TLink>, ILinks<TLink>
    {
        public LinksCombinedConstants<TLink, TLink, int> Constants { get; }
        protected LinksDecoratorBase(ILinks<TLink> links) : base(links) => Constants =
        ↪  links.Constants;
        public virtual TLink Count(IList<TLink> restriction) => Links.Count(restriction);
        public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
        ↪  => Links.Each(handler, restrictions);
        public virtual TLink Create() => Links.Create();
        public virtual TLink Update(IList<TLink> restrictions) => Links.Update(restrictions);
        public virtual void Delete(TLink link) => Links.Delete(link);
    }
}
```

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

```csharp
using System;
using System.Collections.Generic;
using Platform.Disposables;
using Platform.Data.Constants;

namespace Platform.Data.Doublets.Decorators
{
    public abstract class LinksDisposableDecoratorBase<TLink> : DisposableBase, ILinks<TLink>
    {
        public LinksCombinedConstants<TLink, TLink, int> Constants { get; }

        public ILinks<TLink> Links { get; }

        protected LinksDisposableDecoratorBase(ILinks<TLink> links)
        {
            Links = links;
            Constants = links.Constants;
        }

        public virtual TLink Count(IList<TLink> restriction) => Links.Count(restriction);
```

```
22     public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
  ↪      => Links.Each(handler, restrictions);
23
24     public virtual TLink Create() => Links.Create();
25
26     public virtual TLink Update(IList<TLink> restrictions) => Links.Update(restrictions);
27
28     public virtual void Delete(TLink link) => Links.Delete(link);
29
30     protected override bool AllowMultipleDisposeCalls => true;
31
32     protected override void Dispose(bool manual, bool wasDisposed)
33     {
34         if (!wasDisposed)
35         {
36             Links.DisposeIfPossible();
37         }
38     }
39   }
40 }
```

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

```
 1 using System;
 2 using System.Collections.Generic;
 3
 4 namespace Platform.Data.Doublets.Decorators
 5 {
 6     // TODO: Make LinksExternalReferenceValidator. A layer that checks each link to exist or to
  ↪   be external (hybrid link's raw number).
 7     public class LinksInnerReferenceExistenceValidator<TLink> : LinksDecoratorBase<TLink>
 8     {
 9         public LinksInnerReferenceExistenceValidator(ILinks<TLink> links) : base(links) { }
10
11         public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
12         {
13             Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
14             return Links.Each(handler, restrictions);
15         }
16
17         public override TLink Count(IList<TLink> restriction)
18         {
19             Links.EnsureInnerReferenceExists(restriction, nameof(restriction));
20             return Links.Count(restriction);
21         }
22
23         public override TLink Update(IList<TLink> restrictions)
24         {
25             // TODO: Possible values: null, ExistentLink or NonExistentHybrid(ExternalReference)
26             Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
27             return Links.Update(restrictions);
28         }
29
30         public override void Delete(TLink link)
31         {
32             Links.EnsureLinkExists(link, nameof(link));
33             Links.Delete(link);
34         }
35     }
36 }
```

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

```
 1 using System;
 2 using System.Collections.Generic;
 3
 4 namespace Platform.Data.Doublets.Decorators
 5 {
 6     public class LinksItselfConstantToSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
 7     {
 8         private static readonly EqualityComparer<TLink> _equalityComparer =
  ↪       EqualityComparer<TLink>.Default;
 9
10         public LinksItselfConstantToSelfReferenceResolver(ILinks<TLink> links) : base(links) { }
11
12         public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
13         {
14             var constants = Constants;
15             var itselfConstant = constants.Itself;
16             var indexPartConstant = constants.IndexPart;
17             var sourcePartConstant = constants.SourcePart;
```

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

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

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

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

```
1  using System.Collections.Generic;
2
3  namespace Platform.Data.Doublets.Decorators
4  {
5      public class LinksNullConstantToSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
6      {
7          public LinksNullConstantToSelfReferenceResolver(ILinks<TLink> links) : base(links) { }
8
9          public override TLink Create()
10         {
11             var link = Links.Create();
12             return Links.Update(link, link, link);
13         }
14
15         public override TLink Update(IList<TLink> restrictions) =>
            ↪ Links.Update(Links.ResolveConstantAsSelfReference(Constants.Null, restrictions));
16     }
17 }
```

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

```
1  using System.Collections.Generic;
2
3  namespace Platform.Data.Doublets.Decorators
4  {
5      public class LinksUniquenessResolver<TLink> : LinksDecoratorBase<TLink>
6      {
7          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪ EqualityComparer<TLink>.Default;
8
9          public LinksUniquenessResolver(ILinks<TLink> links) : base(links) { }
```

```
10
11          public override TLink Update(IList<TLink> restrictions)
12          {
13              var newLinkAddress = Links.SearchOrDefault(restrictions[Constants.SourcePart],
   ↪   restrictions[Constants.TargetPart]);
14              if (_equalityComparer.Equals(newLinkAddress, default))
15              {
16                  return Links.Update(restrictions);
17              }
18              return ResolveAddressChangeConflict(restrictions[Constants.IndexPart],
   ↪   newLinkAddress);
19          }
20
21          protected virtual TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
   ↪   newLinkAddress)
22          {
23              if (!_equalityComparer.Equals(oldLinkAddress, newLinkAddress) &&
   ↪   Links.Exists(oldLinkAddress))
24              {
25                  Delete(oldLinkAddress);
26              }
27              return newLinkAddress;
28          }
29      }
30  }
```

./Platform.Data.Doublets/Decorators/LinksUniquenessValidator.cs
```
1   using System.Collections.Generic;
2
3   namespace Platform.Data.Doublets.Decorators
4   {
5       public class LinksUniquenessValidator<TLink> : LinksDecoratorBase<TLink>
6       {
7           public LinksUniquenessValidator(ILinks<TLink> links) : base(links) { }
8
9           public override TLink Update(IList<TLink> restrictions)
10          {
11              Links.EnsureDoesNotExists(restrictions[Constants.SourcePart],
   ↪   restrictions[Constants.TargetPart]);
12              return Links.Update(restrictions);
13          }
14      }
15  }
```

./Platform.Data.Doublets/Decorators/LinksUsagesValidator.cs
```
1   using System.Collections.Generic;
2
3   namespace Platform.Data.Doublets.Decorators
4   {
5       public class LinksUsagesValidator<TLink> : LinksDecoratorBase<TLink>
6       {
7           public LinksUsagesValidator(ILinks<TLink> links) : base(links) { }
8
9           public override TLink Update(IList<TLink> restrictions)
10          {
11              Links.EnsureNoUsages(restrictions[Constants.IndexPart]);
12              return Links.Update(restrictions);
13          }
14
15          public override void Delete(TLink link)
16          {
17              Links.EnsureNoUsages(link);
18              Links.Delete(link);
19          }
20      }
21  }
```

./Platform.Data.Doublets/Decorators/NonNullContentsLinkDeletionResolver.cs
```
1   namespace Platform.Data.Doublets.Decorators
2   {
3       public class NonNullContentsLinkDeletionResolver<TLink> : LinksDecoratorBase<TLink>
4       {
5           public NonNullContentsLinkDeletionResolver(ILinks<TLink> links) : base(links) { }
6
7           public override void Delete(TLink linkIndex)
8           {
9               Links.EnforceResetValues(linkIndex);
10              Links.Delete(linkIndex);
```

```csharp
                }
            }
        }
    }
```

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

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

namespace Platform.Data.Doublets.Decorators
{
    /// <summary>
    /// Представляет объект для работы с базой данных (файлом) в формате Links (массива связей).
    /// </summary>
    /// <remarks>
    /// Возможные оптимизации:
    /// Объединение в одном поле Source и Target с уменьшением до 32 бит.
    ///     + меньше объём БД
    ///     - меньше производительность
    ///     - больше ограничение на количество связей в БД)
    /// Ленивое хранение размеров поддеревьев (расчитываемое по мере использования БД)
    ///     + меньше объём БД
    ///     - больше сложность
    ///
    /// 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
    /// Желательно реализовать поддержку переключения между деревьями и битовыми индексами
    ///     (битовыми строками) - вариант матрицы (выстраеваемой лениво).
    ///
    /// Решить отключать ли проверки при компиляции под Release. Т.е. исключения будут
    ///     выбрасываться только при #if DEBUG
    /// </remarks>
    public class UInt64Links : LinksDisposableDecoratorBase<ulong>
    {
        public UInt64Links(ILinks<ulong> links) : base(links) { }

        public override ulong Each(Func<IList<ulong>, ulong> handler, IList<ulong> restrictions)
        {
            this.EnsureLinkIsAnyOrExists(restrictions);
            return Links.Each(handler, restrictions);
        }

        public override ulong Create() => Links.CreatePoint();

        public override ulong Update(IList<ulong> restrictions)
        {
            var constants = Constants;
            var nullConstant = constants.Null;
            if (restrictions.IsNullOrEmpty())
            {
                return nullConstant;
            }
            // TODO: Looks like this is a common type of exceptions linked with restrictions
            //  support
            if (restrictions.Count != 3)
            {
                throw new NotSupportedException();
            }
            var indexPartConstant = constants.IndexPart;
            var updatedLink = restrictions[indexPartConstant];
            this.EnsureLinkExists(updatedLink,
                $"{nameof(restrictions)}[{nameof(indexPartConstant)}]");
            var sourcePartConstant = constants.SourcePart;
            var newSource = restrictions[sourcePartConstant];
            this.EnsureLinkIsItselfOrExists(newSource,
                $"{nameof(restrictions)}[{nameof(sourcePartConstant)}]");
            var targetPartConstant = constants.TargetPart;
            var newTarget = restrictions[targetPartConstant];
            this.EnsureLinkIsItselfOrExists(newTarget,
                $"{nameof(restrictions)}[{nameof(targetPartConstant)}]");
            var existedLink = nullConstant;
            var itselfConstant = constants.Itself;
            if (newSource != itselfConstant && newTarget != itselfConstant)
            {
                existedLink = this.SearchOrDefault(newSource, newTarget);
            }
            if (existedLink == nullConstant)
            {
```

```
67                      var before = Links.GetLink(updatedLink);
68                      if (before[sourcePartConstant] != newSource || before[targetPartConstant] !=
   ↪    newTarget)
69                      {
70                          Links.Update(updatedLink, newSource == itselfConstant ? updatedLink :
   ↪    newSource,
71                                                     newTarget == itselfConstant ? updatedLink :
   ↪    newTarget);
72                      }
73                      return updatedLink;
74                  }
75                  else
76                  {
77                      return this.MergeAndDelete(updatedLink, existedLink);
78                  }
79              }
80
81          public override void Delete(ulong linkIndex)
82          {
83              Links.EnsureLinkExists(linkIndex);
84              Links.EnforceResetValues(linkIndex);
85              this.DeleteAllUsages(linkIndex);
86              Links.Delete(linkIndex);
87          }
88      }
89  }
```

./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   namespace Platform.Data.Doublets.Decorators
10  {
11      /// <remarks>
12      /// What does empty pattern (for condition or substitution) mean? Nothing or Everything?
13      /// 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).
14      ///
15      /// 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.)
16      /// </remarks>
17      internal class UniLinks<TLink> : LinksDecoratorBase<TLink>, IUniLinks<TLink>
18      {
19          private static readonly EqualityComparer<TLink> _equalityComparer =
   ↪    EqualityComparer<TLink>.Default;
20
21          public UniLinks(ILinks<TLink> links) : base(links) { }
22
23          private struct Transition
24          {
25              public IList<TLink> Before;
26              public IList<TLink> After;
27
28              public Transition(IList<TLink> before, IList<TLink> after)
29              {
30                  Before = before;
31                  After = after;
32              }
33          }
34
35          //public static readonly TLink NullConstant = Use<LinksCombinedConstants<TLink, TLink,
   ↪    int>>.Single.Null;
36          //public static readonly IReadOnlyList<TLink> NullLink = new
   ↪    ReadOnlyCollection<TLink>(new List<TLink> { NullConstant, NullConstant, NullConstant
   ↪    });
37
38          // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution
   ↪    (Links-Expression)
39          public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>, TLink>
   ↪    matchedHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
   ↪    substitutedHandler)
40          {
41              ////List<Transition> transitions = null;
```

```
42              ////if (!restriction.IsNullOrEmpty())
43              ////{
44              ////     // Есть причина делать проход (чтение)
45              ////     if (matchedHandler != null)
46              ////     {
47              ////         if (!substitution.IsNullOrEmpty())
48              ////         {
49              ////             // restriction => { 0, 0, 0 } | { 0 } // Create
50              ////             // substitution => { itself, 0, 0 } | { itself, itself, itself } //
    ↪  Create / Update
51              ////             // substitution => { 0, 0, 0 } | { 0 } // Delete
52              ////             transitions = new List<Transition>();
53              ////             if (Equals(substitution[Constants.IndexPart], Constants.Null))
54              ////             {
55              ////                 // If index is Null, that means we always ignore every other
    ↪  value (they are also Null by definition)
56              ////                 var matchDecision = matchedHandler(, NullLink);
57              ////                 if (Equals(matchDecision, Constants.Break))
58              ////                     return false;
59              ////                 if (!Equals(matchDecision, Constants.Skip))
60              ////                     transitions.Add(new Transition(matchedLink, newValue));
61              ////             }
62              ////             else
63              ////             {
64              ////                 Func<T, bool> handler;
65              ////                 handler = link =>
66              ////                 {
67              ////                     var matchedLink = Memory.GetLinkValue(link);
68              ////                     var newValue = Memory.GetLinkValue(link);
69              ////                     newValue[Constants.IndexPart] = Constants.Itself;
70              ////                     newValue[Constants.SourcePart] =
    ↪  Equals(substitution[Constants.SourcePart], Constants.Itself) ?
    ↪  matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
71              ////                     newValue[Constants.TargetPart] =
    ↪  Equals(substitution[Constants.TargetPart], Constants.Itself) ?
    ↪  matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
72              ////                     var matchDecision = matchedHandler(matchedLink, newValue);
73              ////                     if (Equals(matchDecision, Constants.Break))
74              ////                         return false;
75              ////                     if (!Equals(matchDecision, Constants.Skip))
76              ////                         transitions.Add(new Transition(matchedLink, newValue));
77              ////                     return true;
78              ////                 };
79              ////                 if (!Memory.Each(handler, restriction))
80              ////                     return Constants.Break;
81              ////             }
82              ////         }
83              ////         else
84              ////         {
85              ////             Func<T, bool> handler = link =>
86              ////             {
87              ////                 var matchedLink = Memory.GetLinkValue(link);
88              ////                 var matchDecision = matchedHandler(matchedLink, matchedLink);
89              ////                 return !Equals(matchDecision, Constants.Break);
90              ////             };
91              ////             if (!Memory.Each(handler, restriction))
92              ////                 return Constants.Break;
93              ////         }
94              ////     }
95              ////     else
96              ////     {
97              ////         if (substitution != null)
98              ////         {
99              ////             transitions = new List<IList<T>>();
100             ////             Func<T, bool> handler = link =>
101             ////             {
102             ////                 var matchedLink = Memory.GetLinkValue(link);
103             ////                 transitions.Add(matchedLink);
104             ////                 return true;
105             ////             };
106             ////             if (!Memory.Each(handler, restriction))
107             ////                 return Constants.Break;
108             ////         }
109             ////         else
110             ////         {
111             ////             return Constants.Continue;
112             ////         }
```

```
////    }
////}
////if (substitution != null)
////{
////    // Есть причина делать замену (запись)
////    if (substitutedHandler != null)
////    {
////    }
////    else
////    {
////    }
////}
////return Constants.Continue;

//if (restriction.IsNullOrEmpty()) // Create
//{
//    substitution[Constants.IndexPart] = Memory.AllocateLink();
//    Memory.SetLinkValue(substitution);
//}
//else if (substitution.IsNullOrEmpty()) // Delete
//{
//    Memory.FreeLink(restriction[Constants.IndexPart]);
//}
//else if (restriction.EqualTo(substitution)) // Read or ("repeat" the state) // Each
//{
//    // No need to collect links to list
//    // Skip == Continue
//    // No need to check substituedHandler
//    if (!Memory.Each(link => !Equals(matchedHandler(Memory.GetLinkValue(link)),
//  Constants.Break), restriction))
//        return Constants.Break;
//}
//else // Update
//{
//    //List<IList<T>> matchedLinks = null;
//    if (matchedHandler != null)
//    {
//        matchedLinks = new List<IList<T>>();
//        Func<T, bool> handler = link =>
//        {
//            var matchedLink = Memory.GetLinkValue(link);
//            var matchDecision = matchedHandler(matchedLink);
//            if (Equals(matchDecision, Constants.Break))
//                return false;
//            if (!Equals(matchDecision, Constants.Skip))
//                matchedLinks.Add(matchedLink);
//            return true;
//        };
//        if (!Memory.Each(handler, restriction))
//            return Constants.Break;
//    }
//    if (!matchedLinks.IsNullOrEmpty())
//    {
//        var totalMatchedLinks = matchedLinks.Count;
//        for (var i = 0; i < totalMatchedLinks; i++)
//        {
//            var matchedLink = matchedLinks[i];
//            if (substitutedHandler != null)
//            {
//                var newValue = new List<T>(); // TODO: Prepare value to update here
//                // TODO: Decide is it actually needed to use Before and After
//  substitution handling.
//                var substitutedDecision = substitutedHandler(matchedLink,
//  newValue);
//                if (Equals(substitutedDecision, Constants.Break))
//                    return Constants.Break;
//                if (Equals(substitutedDecision, Constants.Continue))
//                {
//                    // Actual update here
//                    Memory.SetLinkValue(newValue);
//                }
//                if (Equals(substitutedDecision, Constants.Skip))
//                {
//                    // Cancel the update. TODO: decide use separate Cancel
//  constant or Skip is enough?
//                }
//            }
```

```csharp
186            //            }
187            //        }
188            //}
189            return Constants.Continue;
190        }
191
192        public TLink Trigger(IList<TLink> patternOrCondition, Func<IList<TLink>, TLink>
            matchHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
            substitutionHandler)
193        {
194            if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
195            {
196                return Constants.Continue;
197            }
198            else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO:
                Check if it is a correct condition
199            {
200                // Or it only applies to trigger without matchHandler.
201                throw new NotImplementedException();
202            }
203            else if (!substitution.IsNullOrEmpty()) // Creation
204            {
205                var before = ArrayPool<TLink>.Empty;
206                // Что должно означать False здесь? Остановиться (перестать идти) или пропустить
                    (пройти мимо) или пустить (взять)?
207                if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                    Constants.Break))
208                {
209                    return Constants.Break;
210                }
211                var after = (IList<TLink>)substitution.ToArray();
212                if (_equalityComparer.Equals(after[0], default))
213                {
214                    var newLink = Links.Create();
215                    after[0] = newLink;
216                }
217                if (substitution.Count == 1)
218                {
219                    after = Links.GetLink(substitution[0]);
220                }
221                else if (substitution.Count == 3)
222                {
223                    Links.Update(after);
224                }
225                else
226                {
227                    throw new NotSupportedException();
228                }
229                if (matchHandler != null)
230                {
231                    return substitutionHandler(before, after);
232                }
233                return Constants.Continue;
234            }
235            else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
236            {
237                if (patternOrCondition.Count == 1)
238                {
239                    var linkToDelete = patternOrCondition[0];
240                    var before = Links.GetLink(linkToDelete);
241                    if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                        Constants.Break))
242                    {
243                        return Constants.Break;
244                    }
245                    var after = ArrayPool<TLink>.Empty;
246                    Links.Update(linkToDelete, Constants.Null, Constants.Null);
247                    Links.Delete(linkToDelete);
248                    if (matchHandler != null)
249                    {
250                        return substitutionHandler(before, after);
251                    }
252                    return Constants.Continue;
253                }
254                else
255                {
256                    throw new NotSupportedException();
257                }
```

```csharp
                    }
                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>
        public IList<IList<IList<TLink>>> Trigger(IList<TLink> condition, IList<TLink>
        ↪ substitution)
        {
            var changes = new List<IList<IList<TLink>>>();
            Trigger(condition, AlwaysContinue, substitution, (before, after) =>
            {
                var change = new[] { before, after };
                changes.Add(change);
                return Constants.Continue;
            });
            return changes;
        }

        private TLink AlwaysContinue(IList<TLink> linkToMatch) => Constants.Continue;
    }
}
```

./Platform.Data.Doublets/DoubletComparer.cs
```csharp
using System.Collections.Generic;
using System.Runtime.CompilerServices;

```

```csharp
namespace Platform.Data.Doublets
{
    /// <remarks>
    /// TODO: Может стоит попробовать ref во всех методах (IRefEqualityComparer)
    /// 2x faster with comparer
    /// </remarks>
    public class DoubletComparer<T> : IEqualityComparer<Doublet<T>>
    {
        public static readonly DoubletComparer<T> Default = new DoubletComparer<T>();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Equals(Doublet<T> x, Doublet<T> y) => x.Equals(y);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public int GetHashCode(Doublet<T> obj) => obj.GetHashCode();
    }
}
```

./Platform.Data.Doublets/Doublet.cs
```csharp
using System;
using System.Collections.Generic;

namespace Platform.Data.Doublets
{
    public struct Doublet<T> : IEquatable<Doublet<T>>
    {
        private static readonly EqualityComparer<T> _equalityComparer =
            EqualityComparer<T>.Default;

        public T Source { get; set; }
        public T Target { get; set; }

        public Doublet(T source, T target)
        {
            Source = source;
            Target = target;
        }

        public override string ToString() => $"{Source}->{Target}";

        public bool Equals(Doublet<T> other) => _equalityComparer.Equals(Source, other.Source)
            && _equalityComparer.Equals(Target, other.Target);

        public override bool Equals(object obj) => obj is Doublet<T> doublet ?
            base.Equals(doublet) : false;

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

./Platform.Data.Doublets/Hybrid.cs
```csharp
using System;
using System.Reflection;
using Platform.Reflection;
using Platform.Converters;
using Platform.Exceptions;

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 => 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);
```

```
29        var abs = typeof(Numbers.Math).GetTypeInfo().GetMethod("Abs").MakeGenericMethod(sign⌋
          ↪  edType);
30        var negate = typeof(Numbers.Math).GetTypeInfo().GetMethod("Negate").MakeGenericMetho⌋
          ↪  d(signedType);
31        var absoluteValue = abs.Invoke(null, new[] { signedValue });
32        var resultValue = isExternal ? negate.Invoke(null, new[] { absoluteValue }) :
          ↪  absoluteValue;
33        Value = To.UnsignedAs<T>(resultValue);
34    }
35
36    public static implicit operator Hybrid<T>(T integer) => new Hybrid<T>(integer);
37
38    public static explicit operator Hybrid<T>(ulong integer) => new Hybrid<T>(integer);
39
40    public static explicit operator Hybrid<T>(long integer) => new Hybrid<T>(integer);
41
42    public static explicit operator Hybrid<T>(uint integer) => new Hybrid<T>(integer);
43
44    public static explicit operator Hybrid<T>(int integer) => new Hybrid<T>(integer);
45
46    public static explicit operator Hybrid<T>(ushort integer) => new Hybrid<T>(integer);
47
48    public static explicit operator Hybrid<T>(short integer) => new Hybrid<T>(integer);
49
50    public static explicit operator Hybrid<T>(byte integer) => new Hybrid<T>(integer);
51
52    public static explicit operator Hybrid<T>(sbyte integer) => new Hybrid<T>(integer);
53
54    public static implicit operator T(Hybrid<T> hybrid) => hybrid.Value;
55
56    public static explicit operator ulong(Hybrid<T> hybrid) =>
          ↪  Convert.ToUInt64(hybrid.Value);
57
58    public static explicit operator long(Hybrid<T> hybrid) => hybrid.AbsoluteValue;
59
60    public static explicit operator uint(Hybrid<T> hybrid) => Convert.ToUInt32(hybrid.Value);
61
62    public static explicit operator int(Hybrid<T> hybrid) =>
          ↪  Convert.ToInt32(hybrid.AbsoluteValue);
63
64    public static explicit operator ushort(Hybrid<T> hybrid) =>
          ↪  Convert.ToUInt16(hybrid.Value);
65
66    public static explicit operator short(Hybrid<T> hybrid) =>
          ↪  Convert.ToInt16(hybrid.AbsoluteValue);
67
68    public static explicit operator byte(Hybrid<T> hybrid) => Convert.ToByte(hybrid.Value);
69
70    public static explicit operator sbyte(Hybrid<T> hybrid) =>
          ↪  Convert.ToSByte(hybrid.AbsoluteValue);
71
72    public override string ToString() => IsNothing ? default(T) == null ? "Nothing" :
          ↪  default(T).ToString() : IsExternal ? $"<{AbsoluteValue}>" : Value.ToString();
73    }
74 }
```

./Platform.Data.Doublets/ILinks.cs

```
1  using Platform.Data.Constants;
2
3  namespace Platform.Data.Doublets
4  {
5      public interface ILinks<TLink> : ILinks<TLink, LinksCombinedConstants<TLink, TLink, int>>
6      {
7      }
8  }
```

./Platform.Data.Doublets/ILinksExtensions.cs

```
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System.Linq;
5  using System.Runtime.CompilerServices;
6  using Platform.Ranges;
7  using Platform.Collections.Arrays;
8  using Platform.Numbers;
9  using Platform.Random;
10 using Platform.Setters;
11 using Platform.Data.Exceptions;
12
13 namespace Platform.Data.Doublets
```

```csharp
{
    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>
        /// 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;
            });
        }
    }
}
```

```csharp
            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))
                {
                    //throw new Exception(string.Format("Невозможно выбрать путь, так как и
                    //    Source и Target совпадают с элементом пути {0}.", next));
                    return false;
                }
                if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
                    target))
                {
                    //throw new Exception(string.Format("Невозможно продолжить путь через
                    //    элемент пути {0}", next));
                    return false;
                }
                current = next;
            }
            return true;
        }

        /// <remarks>
        /// Может потребовать дополнительного стека для PathElement's при использовании
        ///    SequenceWalker.
        /// </remarks>
        public static TLink GetByKeys<TLink>(this ILinks<TLink> links, TLink root, params int[]
            path)
        {
            links.EnsureLinkExists(root, "root");
            var currentLink = root;
            for (var i = 0; i < path.Length; i++)
            {
                currentLink = links.GetLink(currentLink)[path[i]];
            }
            return currentLink;
        }

        public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this ILinks<TLink>
            links, TLink root, ulong size, ulong index)
```

```csharp
156         {
157             var constants = links.Constants;
158             var source = constants.SourcePart;
159             var target = constants.TargetPart;
160             if (!Numbers.Math.IsPowerOfTwo(size))
161             {
162                 throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes other
    ↪        than powers of two are not supported.");
163             }
164             var path = new BitArray(BitConverter.GetBytes(index));
165             var length = Bit.GetLowestPosition(size);
166             links.EnsureLinkExists(root, "root");
167             var currentLink = root;
168             for (var i = length - 1; i >= 0; i--)
169             {
170                 currentLink = links.GetLink(currentLink)[path[i] ? target : source];
171             }
172             return currentLink;
173         }
174
175         #endregion
176
177         /// <summary>
178         /// Возвращает индекс указанной связи.
179         /// </summary>
180         /// <param name="links">Хранилище связей.</param>
181         /// <param name="link">Связь представленная списком, состоящим из её адреса и
    ↪    содержимого.</param>
182         /// <returns>Индекс начальной связи для указанной связи.</returns>
183         [MethodImpl(MethodImplOptions.AggressiveInlining)]
184         public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
    ↪    link[links.Constants.IndexPart];
185
186         /// <summary>
187         /// Возвращает индекс начальной (Source) связи для указанной связи.
188         /// </summary>
189         /// <param name="links">Хранилище связей.</param>
190         /// <param name="link">Индекс связи.</param>
191         /// <returns>Индекс начальной связи для указанной связи.</returns>
192         [MethodImpl(MethodImplOptions.AggressiveInlining)]
193         public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink link) =>
    ↪    links.GetLink(link)[links.Constants.SourcePart];
194
195         /// <summary>
196         /// Возвращает индекс начальной (Source) связи для указанной связи.
197         /// </summary>
198         /// <param name="links">Хранилище связей.</param>
199         /// <param name="link">Связь представленная списком, состоящим из её адреса и
    ↪    содержимого.</param>
200         /// <returns>Индекс начальной связи для указанной связи.</returns>
201         [MethodImpl(MethodImplOptions.AggressiveInlining)]
202         public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
    ↪    link[links.Constants.SourcePart];
203
204         /// <summary>
205         /// Возвращает индекс конечной (Target) связи для указанной связи.
206         /// </summary>
207         /// <param name="links">Хранилище связей.</param>
208         /// <param name="link">Индекс связи.</param>
209         /// <returns>Индекс конечной связи для указанной связи.</returns>
210         [MethodImpl(MethodImplOptions.AggressiveInlining)]
211         public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
    ↪    links.GetLink(link)[links.Constants.TargetPart];
212
213         /// <summary>
214         /// Возвращает индекс конечной (Target) связи для указанной связи.
215         /// </summary>
216         /// <param name="links">Хранилище связей.</param>
217         /// <param name="link">Связь представленная списком, состоящим из её адреса и
    ↪    содержимого.</param>
218         /// <returns>Индекс конечной связи для указанной связи.</returns>
219         [MethodImpl(MethodImplOptions.AggressiveInlining)]
220         public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
    ↪    link[links.Constants.TargetPart];
221
222         /// <summary>
223         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
    ↪    (handler) для каждой подходящей связи.
```

```csharp
224          /// </summary>
225          /// <param name="links">Хранилище связей.</param>
226          /// <param name="handler">Обработчик каждой подходящей связи.</param>
227          /// <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>
228          /// <returns>True, в случае если проход по связям не был прерван и False в обратном
             ↪ случае.</returns>
229          [MethodImpl(MethodImplOptions.AggressiveInlining)]
230          public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
             ↪ handler, params TLink[] restrictions)
231              => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
                 ↪ links.Constants.Continue);
232
233          /// <summary>
234          /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
             ↪ (handler) для каждой подходящей связи.
235          /// </summary>
236          /// <param name="links">Хранилище связей.</param>
237          /// <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>
238          /// <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>
239          /// <param name="handler">Обработчик каждой подходящей связи.</param>
240          /// <returns>True, в случае если проход по связям не был прерван и False в обратном
             ↪ случае.</returns>
241          [MethodImpl(MethodImplOptions.AggressiveInlining)]
242          public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
             ↪ Func<TLink, bool> handler)
243          {
244              var constants = links.Constants;
245              return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
                 ↪ constants.Break, constants.Any, source, target);
246          }
247
248          /// <summary>
249          /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
             ↪ (handler) для каждой подходящей связи.
250          /// </summary>
251          /// <param name="links">Хранилище связей.</param>
252          /// <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>
253          /// <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>
254          /// <param name="handler">Обработчик каждой подходящей связи.</param>
255          /// <returns>True, в случае если проход по связям не был прерван и False в обратном
             ↪ случае.</returns>
256          [MethodImpl(MethodImplOptions.AggressiveInlining)]
257          public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
             ↪ Func<IList<TLink>, TLink> handler)
258          {
259              var constants = links.Constants;
260              return links.Each(handler, constants.Any, source, target);
261          }
262
263          [MethodImpl(MethodImplOptions.AggressiveInlining)]
264          public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
             ↪ restrictions)
265          {
266              long arraySize = (Integer<TLink>)links.Count(restrictions);
267              var array = new IList<TLink>[arraySize];
268              if (arraySize > 0)
269              {
270                  var filler = new ArrayFiller<IList<TLink>, TLink>(array,
                     ↪ links.Constants.Continue);
271                  links.Each(filler.AddAndReturnConstant, restrictions);
272              }
273              return array;
274          }
275
276          [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
        public static IList<TLink> AllIndices<TLink>(this ILinks<TLink> links, params TLink[]
        ↪  restrictions)
        {
            long arraySize = (Integer<TLink>)links.Count(restrictions);
            var array = new TLink[arraySize];
            if (arraySize > 0)
            {
                var filler = new ArrayFiller<TLink, TLink>(array, links.Constants.Continue);
                links.Each(filler.AddFirstAndReturnConstant, restrictions));
            }
            return array;
        }

        /// <summary>
        /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
        ↪  в хранилище связей.
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="source">Начало связи.</param>
        /// <param name="target">Конец связи.</param>
        /// <returns>Значение, определяющее существует ли связь.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;

        #region Ensure
        // TODO: May be move to EnsureExtensions or make it both there and here

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
        ↪  reference, string argumentName)
        {
            if (links.IsInnerReference(reference) && !links.Exists(reference))
            {
                throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
        ↪  IList<TLink> restrictions, string argumentName)
        {
            for (int i = 0; i < restrictions.Count; i++)
            {
                links.EnsureInnerReferenceExists(restrictions[i], argumentName);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
        ↪  restrictions)
        {
            for (int i = 0; i < restrictions.Count; i++)
            {
                links.EnsureLinkIsAnyOrExists(restrictions[i], nameof(restrictions));
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
        ↪  string argumentName)
        {
            var equalityComparer = EqualityComparer<TLink>.Default;
            if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
            {
                throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links, TLink
        ↪  link, string argumentName)
        {
            var equalityComparer = EqualityComparer<TLink>.Default;
            if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
            {
                throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
```

```csharp
            }
        }

        /// <param name="links">Хранилище связей.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink source,
        ↪ TLink target)
        {
            if (links.Exists(source, target))
            {
                throw new LinkWithSameValueAlreadyExistsException();
            }
        }

        /// <param name="links">Хранилище связей.</param>
        public static void EnsureNoUsages<TLink>(this ILinks<TLink> links, TLink link)
        {
            if (links.HasUsages(link))
            {
                throw new ArgumentLinkHasDependenciesException<TLink>(link);
            }
        }

        /// <param name="links">Хранилище связей.</param>
        public static void EnsureCreated<TLink>(this ILinks<TLink> links, params TLink[]
        ↪ addresses) => links.EnsureCreated(links.Create, addresses);

        /// <param name="links">Хранилище связей.</param>
        public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params TLink[]
        ↪ addresses) => links.EnsureCreated(links.CreatePoint, addresses);

        /// <param name="links">Хранилище связей.</param>
        public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
        ↪ params TLink[] addresses)
        {
            var constants = links.Constants;
            var nonExistentAddresses = new HashSet<ulong>(addresses.Where(x =>
            ↪ !links.Exists(x)).Select(x => (ulong)(Integer<TLink>)x));
            if (nonExistentAddresses.Count > 0)
            {
                var max = nonExistentAddresses.Max();
                // TODO: Эту верхнюю границу нужно разрешить переопределять (проверить
                ↪ применяется ли эта логика)
                max = System.Math.Min(max, (Integer<TLink>)constants.MaxPossibleIndex);
                var createdLinks = new List<TLink>();
                var equalityComparer = EqualityComparer<TLink>.Default;
                TLink createdLink = creator();
                while (!equalityComparer.Equals(createdLink, (Integer<TLink>)max))
                {
                    createdLinks.Add(createdLink);
                }
                for (var i = 0; i < createdLinks.Count; i++)
                {
                    if (!nonExistentAddresses.Contains((Integer<TLink>)createdLinks[i]))
                    {
                        links.Delete(createdLinks[i]);
                    }
                }
            }
        }

        #endregion

        /// <param name="links">Хранилище связей.</param>
        public static ulong CountUsages<TLink>(this ILinks<TLink> links, TLink link)
        {
            var constants = links.Constants;
            var values = links.GetLink(link);
            ulong usagesAsSource = (Integer<TLink>)links.Count(new Link<TLink>(constants.Any,
            ↪ link, constants.Any));
            var equalityComparer = EqualityComparer<TLink>.Default;
            if (equalityComparer.Equals(values[constants.SourcePart], link))
            {
                usagesAsSource--;
            }
            ulong usagesAsTarget = (Integer<TLink>)links.Count(new Link<TLink>(constants.Any,
            ↪ constants.Any, link));
            if (equalityComparer.Equals(values[constants.TargetPart], link))
```

```csharp
416                 {
417                     usagesAsTarget--;
418                 }
419                 return usagesAsSource + usagesAsTarget;
420             }
421
422             /// <param name="links">Хранилище связей.</param>
423             [MethodImpl(MethodImplOptions.AggressiveInlining)]
424             public static bool HasUsages<TLink>(this ILinks<TLink> links, TLink link) =>
              ↪   links.CountUsages(link) > 0;
425
426             /// <param name="links">Хранилище связей.</param>
427             [MethodImpl(MethodImplOptions.AggressiveInlining)]
428             public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source,
              ↪   TLink target)
429             {
430                 var constants = links.Constants;
431                 var values = links.GetLink(link);
432                 var equalityComparer = EqualityComparer<TLink>.Default;
433                 return equalityComparer.Equals(values[constants.SourcePart], source) &&
                  ↪   equalityComparer.Equals(values[constants.TargetPart], target);
434             }
435
436             /// <summary>
437             /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
438             /// </summary>
439             /// <param name="links">Хранилище связей.</param>
440             /// <param name="source">Индекс связи, которая является началом для искомой
              ↪   связи.</param>
441             /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
442             /// <returns>Индекс искомой связи с указанными Source (началом) и Target
              ↪   (концом).</returns>
443             [MethodImpl(MethodImplOptions.AggressiveInlining)]
444             public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
              ↪   target)
445             {
446                 var contants = links.Constants;
447                 var setter = new Setter<TLink, TLink>(contants.Continue, contants.Break, default);
448                 links.Each(setter.SetFirstAndReturnFalse, contants.Any, source, target);
449                 return setter.Result;
450             }
451
452             /// <param name="links">Хранилище связей.</param>
453             [MethodImpl(MethodImplOptions.AggressiveInlining)]
454             public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
455             {
456                 var link = links.Create();
457                 return links.Update(link, link, link);
458             }
459
460             /// <param name="links">Хранилище связей.</param>
461             [MethodImpl(MethodImplOptions.AggressiveInlining)]
462             public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
              ↪   target) => links.Update(links.Create(), source, target);
463
464             /// <summary>
465             /// Обновляет связь с указанными началом (Source) и концом (Target)
466             /// на связь с указанными началом (NewSource) и концом (NewTarget).
467             /// </summary>
468             /// <param name="links">Хранилище связей.</param>
469             /// <param name="link">Индекс обновляемой связи.</param>
470             /// <param name="newSource">Индекс связи, которая является началом связи, на которую
              ↪   выполняется обновление.</param>
471             /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
              ↪   выполняется обновление.</param>
472             /// <returns>Индекс обновлённой связи.</returns>
473             [MethodImpl(MethodImplOptions.AggressiveInlining)]
474             public static TLink Update<TLink>(this ILinks<TLink> links, TLink link, TLink newSource,
              ↪   TLink newTarget) => links.Update(new Link<TLink>(link, newSource, newTarget));
475
476             /// <summary>
477             /// Обновляет связь с указанными началом (Source) и концом (Target)
478             /// на связь с указанными началом (NewSource) и концом (NewTarget).
479             /// </summary>
480             /// <param name="links">Хранилище связей.</param>
```

```csharp
        /// <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>
        /// <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;
```

```
545                 var link = links.SearchOrDefault(source, target);
546                 if (equalityComparer.Equals(link, default))
547                 {
548                     return links.CreateAndUpdate(newSource, newTarget);
549                 }
550                 if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
        ↪ target))
551                 {
552                     return link;
553                 }
554                 return links.Update(link, newSource, newTarget);
555             }
556
557             /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
558             /// <param name="links">Хранилище связей.</param>
559             /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
560             /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
561             [MethodImpl(MethodImplOptions.AggressiveInlining)]
562             public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
        ↪ target)
563             {
564                 var link = links.SearchOrDefault(source, target);
565                 if (!EqualityComparer<TLink>.Default.Equals(link, default))
566                 {
567                     links.Delete(link);
568                     return link;
569                 }
570                 return default;
571             }
572
573             /// <summary>Удаляет несколько связей.</summary>
574             /// <param name="links">Хранилище связей.</param>
575             /// <param name="deletedLinks">Список адресов связей к удалению.</param>
576             [MethodImpl(MethodImplOptions.AggressiveInlining)]
577             public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
578             {
579                 for (int i = 0; i < deletedLinks.Count; i++)
580                 {
581                     links.Delete(deletedLinks[i]);
582                 }
583             }
584
585             /// <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>
586             public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
587             {
588                 var anyConstant = links.Constants.Any;
589                 var usagesAsSourceQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
590                 links.DeleteByQuery(usagesAsSourceQuery);
591                 var usagesAsTargetQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
592                 links.DeleteByQuery(usagesAsTargetQuery);
593             }
594
595             public static void DeleteByQuery<TLink>(this ILinks<TLink> links, Link<TLink> query)
596             {
597                 var count = (Integer<TLink>)links.Count(query);
598                 if (count > 0)
599                 {
600                     var queryResult = new TLink[count];
601                     var queryResultFiller = new ArrayFiller<TLink, TLink>(queryResult,
            ↪ links.Constants.Continue);
602                     links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
603                     for (var i = (long)count - 1; i >= 0; i--)
604                     {
605                         links.Delete(queryResult[i]);
606                     }
607                 }
608             }
609
610             // TODO: Move to Platform.Data
611             public static bool AreValuesReset<TLink>(this ILinks<TLink> links, TLink linkIndex)
612             {
613                 var nullConstant = links.Constants.Null;
614                 var equalityComparer = EqualityComparer<TLink>.Default;
615                 var link = links.GetLink(linkIndex);
616                 for (int i = 1; i < link.Count; i++)
617                 {
```

```csharp
                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)
                {
                    var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
                    if (totalUsages > 0)
                    {
                        var usages = ArrayPool.Allocate<TLink>(totalUsages);
                        var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
                        ↪ links.Constants.Continue);
                        var i = 0L;
                        if (usagesAsSourceCount > 0)
                        {
                            links.Each(usagesFiller.AddFirstAndReturnConstant,
                            ↪ usagesAsSourceQuery);
                            for (; i < usagesAsSourceCount; i++)
                            {
                                var usage = usages[i];
                                if (!equalityComparer.Equals(usage, oldLinkIndex))
                                {
                                    links.Update(usage, newLinkIndex, links.GetTarget(usage));
                                }
                            }
                        }
                        if (usagesAsTargetCount > 0)
                        {
                            links.Each(usagesFiller.AddFirstAndReturnConstant,
                            ↪ usagesAsTargetQuery);
                            for (; i < usages.Length; i++)
                            {
                                var usage = usages[i];
                                if (!equalityComparer.Equals(usage, oldLinkIndex))
                                {
                                    links.Update(usage, links.GetSource(usage), newLinkIndex);
```

```
686                            }
687                        }
688                    }
689                    ArrayPool.Free(usages);
690                }
691            }
692        }
693        return newLinkIndex;
694    }

    /// <summary>
    /// Replace one link with another (replaced link is deleted, children are updated or
    ↪  deleted).
    /// </summary>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
    ↪  TLink newLinkIndex)
    {
        var equalityComparer = EqualityComparer<TLink>.Default;
        if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
        {
            links.MergeUsages(oldLinkIndex, newLinkIndex);
            links.Delete(oldLinkIndex);
        }
        return newLinkIndex;
    }
}
```

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

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

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

```
1   using System.Collections.Generic;
2   using Platform.Interfaces;
3
4   namespace Platform.Data.Doublets.Incrementers
5   {
6       public class UnaryNumberIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
7       {
8           private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
9
10          private readonly TLink _unaryOne;
11
```

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

./Platform.Data.Doublets/ISynchronizedLinks.cs

```
1  using Platform.Data.Constants;
2
3  namespace Platform.Data.Doublets
4  {
5      public interface ISynchronizedLinks<TLink> : ISynchronizedLinks<TLink, ILinks<TLink>,
       ↪    LinksCombinedConstants<TLink, TLink, int>>, ILinks<TLink>
6      {
7      }
8  }
```

./Platform.Data.Doublets/Link.cs

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

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

```csharp
            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))
                {
                    return _constants.TargetPart;
                }
                return -1;
            }

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

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

            #endregion
        }
    }
```

./Platform.Data.Doublets/LinkExtensions.cs

```csharp
namespace Platform.Data.Doublets
{
    public static class LinkExtensions
    {
        public static bool IsFullPoint<TLink>(this Link<TLink> link) =>
        ↪  Point<TLink>.IsFullPoint(link);
        public static bool IsPartialPoint<TLink>(this Link<TLink> link) =>
        ↪  Point<TLink>.IsPartialPoint(link);
    }
}
```

./Platform.Data.Doublets/LinksOperatorBase.cs

```csharp
namespace Platform.Data.Doublets
{
    public abstract class LinksOperatorBase<TLink>
    {
        public ILinks<TLink> Links { get; }
        protected LinksOperatorBase(ILinks<TLink> links) => Links = links;
    }
}
```

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

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

```csharp
using Platform.Interfaces;

namespace Platform.Data.Doublets.PropertyOperators
{
    public class PropertiesOperator<TLink> : LinksOperatorBase<TLink>,
     ↪  IPropertiesOperator<TLink, TLink, TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
         ↪  EqualityComparer<TLink>.Default;

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

        public TLink GetValue(TLink @object, TLink property)
        {
            var objectProperty = Links.SearchOrDefault(@object, property);
            if (_equalityComparer.Equals(objectProperty, default))
            {
                return default;
            }
            var valueLink = Links.All(Links.Constants.Any, objectProperty).SingleOrDefault();
            if (valueLink == null)
            {
                return default;
            }
            return Links.GetTarget(valueLink[Links.Constants.IndexPart]);
        }

        public void SetValue(TLink @object, TLink property, TLink value)
        {
            var objectProperty = Links.GetOrCreate(@object, property);
            Links.DeleteMany(Links.AllIndices(Links.Constants.Any, objectProperty));
            Links.GetOrCreate(objectProperty, value);
        }
    }
}
```

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

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

namespace Platform.Data.Doublets.PropertyOperators
{
    public class PropertyOperator<TLink> : LinksOperatorBase<TLink>, IPropertyOperator<TLink,
     ↪  TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
         ↪  EqualityComparer<TLink>.Default;

        private readonly TLink _propertyMarker;
        private readonly TLink _propertyValueMarker;

        public PropertyOperator(ILinks<TLink> links, TLink propertyMarker, TLink
         ↪  propertyValueMarker) : base(links)
        {
            _propertyMarker = propertyMarker;
            _propertyValueMarker = propertyValueMarker;
        }

        public TLink Get(TLink link)
        {
            var property = Links.SearchOrDefault(link, _propertyMarker);
            var container = GetContainer(property);
            var value = GetValue(container);
            return value;
        }

        private TLink GetContainer(TLink property)
        {
            var valueContainer = default(TLink);
            if (_equalityComparer.Equals(property, default))
            {
                return valueContainer;
            }
            var constants = Links.Constants;
            var countinueConstant = constants.Continue;
            var breakConstant = constants.Break;
            var anyConstant = constants.Any;
            var query = new Link<TLink>(anyConstant, property, anyConstant);
            Links.Each(candidate =>
            {
```

```csharp
                    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

// 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();
            public static readonly int RightAsSourceOffset = Marshal.OffsetOf(typeof(Link),
                nameof(RightAsSource)).ToInt32();
```

```csharp
            public static readonly int SizeAsSourceOffset = Marshal.OffsetOf(typeof(Link),
                nameof(SizeAsSource)).ToInt32();
            public static readonly int LeftAsTargetOffset = Marshal.OffsetOf(typeof(Link),
                nameof(LeftAsTarget)).ToInt32();
            public static readonly int RightAsTargetOffset = Marshal.OffsetOf(typeof(Link),
                nameof(RightAsTarget)).ToInt32();
            public static readonly int SizeAsTargetOffset = Marshal.OffsetOf(typeof(Link),
                nameof(SizeAsTarget)).ToInt32();

            public TLink Source;
            public TLink Target;
            public TLink LeftAsSource;
            public TLink RightAsSource;
            public TLink SizeAsSource;
            public TLink LeftAsTarget;
            public TLink RightAsTarget;
            public TLink SizeAsTarget;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetSource(IntPtr pointer) => (pointer +
                SourceOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetTarget(IntPtr pointer) => (pointer +
                TargetOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetLeftAsSource(IntPtr pointer) => (pointer +
                LeftAsSourceOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetRightAsSource(IntPtr pointer) => (pointer +
                RightAsSourceOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetSizeAsSource(IntPtr pointer) => (pointer +
                SizeAsSourceOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetLeftAsTarget(IntPtr pointer) => (pointer +
                LeftAsTargetOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetRightAsTarget(IntPtr pointer) => (pointer +
                RightAsTargetOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetSizeAsTarget(IntPtr pointer) => (pointer +
                SizeAsTargetOffset).GetValue<TLink>();

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetSource(IntPtr pointer, TLink value) => (pointer +
                SourceOffset).SetValue(value);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetTarget(IntPtr pointer, TLink value) => (pointer +
                TargetOffset).SetValue(value);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetLeftAsSource(IntPtr pointer, TLink value) => (pointer +
                LeftAsSourceOffset).SetValue(value);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetRightAsSource(IntPtr pointer, TLink value) => (pointer +
                RightAsSourceOffset).SetValue(value);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetSizeAsSource(IntPtr pointer, TLink value) => (pointer +
                SizeAsSourceOffset).SetValue(value);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetLeftAsTarget(IntPtr pointer, TLink value) => (pointer +
                LeftAsTargetOffset).SetValue(value);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetRightAsTarget(IntPtr pointer, TLink value) => (pointer +
                RightAsTargetOffset).SetValue(value);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetSizeAsTarget(IntPtr pointer, TLink value) => (pointer +
                SizeAsTargetOffset).SetValue(value);
        }

        private struct LinksHeader
        {
            public static readonly int AllocatedLinksOffset =
                Marshal.OffsetOf(typeof(LinksHeader), nameof(AllocatedLinks)).ToInt32();
            public static readonly int ReservedLinksOffset =
                Marshal.OffsetOf(typeof(LinksHeader), nameof(ReservedLinks)).ToInt32();
            public static readonly int FreeLinksOffset = Marshal.OffsetOf(typeof(LinksHeader),
                nameof(FreeLinks)).ToInt32();
```

```csharp
        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;

    private LinksTargetsTreeMethods _targetsTreeMethods;
    private LinksSourcesTreeMethods _sourcesTreeMethods;
```

```csharp
        // 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
        private UnusedLinksListMethods _unusedLinksListMethods;

        /// <summary>
        /// Возвращает общее число связей находящихся в хранилище.
        /// </summary>
        private TLink Total => Subtract(LinksHeader.GetAllocatedLinks(_header),
            LinksHeader.GetFreeLinks(_header));

        public LinksCombinedConstants<TLink, TLink, int> Constants { get; }

        public ResizableDirectMemoryLinks(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 ResizableDirectMemoryLinks(string address, long memoryReservationStep)
            : this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
                memoryReservationStep)
        {
        }

        public ResizableDirectMemoryLinks(IResizableDirectMemory memory)
            : this(memory, DefaultLinksSizeStep)
        {
        }

        public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
            memoryReservationStep)
        {
            Constants = Default<LinksCombinedConstants<TLink, TLink, 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)(Integer<TLink>)LinksHeader.GetAllocatedLinks(_header)
                * LinkSizeInBytes) + LinkHeaderSizeInBytes;
            // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
            LinksHeader.SetReservedLinks(_header, (Integer<TLink>)((_memory.ReservedCapacity -
                LinkHeaderSizeInBytes) / LinkSizeInBytes));
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Count(IList<TLink> 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 (_equalityComparer.Equals(index, Constants.Any))
                {
                    return Total;
                }
                return Exists(index) ? Integer<TLink>.One : Integer<TLink>.Zero;
            }
            if (restrictions.Count == 2)
            {
                var index = restrictions[Constants.IndexPart];
                var value = restrictions[1];
                if (_equalityComparer.Equals(index, Constants.Any))
```

```csharp
                    {
                        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;
                            }
                            return Integer<TLink>.Zero;
                        }
```

```csharp
                        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))
                    {
                        return handler(GetLinkStruct(index));
```

```
371                              }
372                              var storedLinkValue = GetLinkUnsafe(index);
373                              if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
374                                  _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
375                              {
376                                  return handler(GetLinkStruct(index));
377                              }
378                              return Constants.Continue;
379                          }
380                      }
381                  if (restrictions.Count == 3)
382                  {
383                      var index = restrictions[Constants.IndexPart];
384                      var source = restrictions[Constants.SourcePart];
385                      var target = restrictions[Constants.TargetPart];
386                      if (_equalityComparer.Equals(index, Constants.Any))
387                      {
388                          if (_equalityComparer.Equals(source, Constants.Any) &&
                             ↪  _equalityComparer.Equals(target, Constants.Any))
389                          {
390                              return Each(handler, ArrayPool<TLink>.Empty);
391                          }
392                          else if (_equalityComparer.Equals(source, Constants.Any))
393                          {
394                              return _targetsTreeMethods.EachUsage(target, handler);
395                          }
396                          else if (_equalityComparer.Equals(target, Constants.Any))
397                          {
398                              return _sourcesTreeMethods.EachUsage(source, handler);
399                          }
400                          else //if(source != Any && target != Any)
401                          {
402                              var link = _sourcesTreeMethods.Search(source, target);
403                              return _equalityComparer.Equals(link, Constants.Null) ?
                             ↪  Constants.Continue : handler(GetLinkStruct(link));
404                          }
405                      }
406                      else
407                      {
408                          if (!Exists(index))
409                          {
410                              return Constants.Continue;
411                          }
412                          if (_equalityComparer.Equals(source, Constants.Any) &&
                             ↪  _equalityComparer.Equals(target, Constants.Any))
413                          {
414                              return handler(GetLinkStruct(index));
415                          }
416                          var storedLinkValue = GetLinkUnsafe(index);
417                          if (!_equalityComparer.Equals(source, Constants.Any) &&
                             ↪  !_equalityComparer.Equals(target, Constants.Any))
418                          {
419                              if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source) &&
420                                  _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
421                              {
422                                  return handler(GetLinkStruct(index));
423                              }
424                              return Constants.Continue;
425                          }
426                          var value = default(TLink);
427                          if (_equalityComparer.Equals(source, Constants.Any))
428                          {
429                              value = target;
430                          }
431                          if (_equalityComparer.Equals(target, Constants.Any))
432                          {
433                              value = source;
434                          }
435                          if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
436                              _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
437                          {
438                              return handler(GetLinkStruct(index));
439                          }
440                          return Constants.Continue;
441                      }
442                  }
443              throw new NotSupportedException("Другие размеры и способы ограничений не
               ↪  поддерживаются.");
```

```csharp
            }

        /// <remarks>
        /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
        ↪   в другом месте (но не в менеджере памяти, а в логике Links)
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Update(IList<TLink> values)
        {
            var linkIndex = values[Constants.IndexPart];
            var link = GetLinkUnsafe(linkIndex);
            // Будет корректно работать только в том случае, если пространство выделенной связи
            ↪   предварительно заполнено нулями
            if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
            {
                _sourcesTreeMethods.Detach(LinksHeader.GetFirstAsSourcePointer(_header),
                ↪   linkIndex);
            }
            if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
            {
                _targetsTreeMethods.Detach(LinksHeader.GetFirstAsTargetPointer(_header),
                ↪   linkIndex);
            }
            Link.SetSource(link, values[Constants.SourcePart]);
            Link.SetTarget(link, values[Constants.TargetPart]);
            if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
            {
                _sourcesTreeMethods.Attach(LinksHeader.GetFirstAsSourcePointer(_header),
                ↪   linkIndex);
            }
            if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
            {
                _targetsTreeMethods.Attach(LinksHeader.GetFirstAsTargetPointer(_header),
                ↪   linkIndex);
            }
            return linkIndex;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Link<TLink> GetLinkStruct(TLink linkIndex)
        {
            var link = GetLinkUnsafe(linkIndex);
            return new Link<TLink>(linkIndex, Link.GetSource(link), Link.GetTarget(link));
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private IntPtr GetLinkUnsafe(TLink linkIndex) => _links.GetElement(LinkSizeInBytes,
        ↪   linkIndex);

        /// <remarks>
        /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
        ↪   пространство
        /// </remarks>
        public TLink Create()
        {
            var freeLink = LinksHeader.GetFirstFreeLink(_header);
            if (!_equalityComparer.Equals(freeLink, Constants.Null))
            {
                _unusedLinksListMethods.Detach(freeLink);
            }
            else
            {
                if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
                ↪   Constants.MaxPossibleIndex) > 0)
                {
                    throw new
                    ↪   LinksLimitReachedException((Integer<TLink>)Constants.MaxPossibleIndex);
                }
                if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
                ↪   Decrement(LinksHeader.GetReservedLinks(_header))) >= 0)
                {
                    _memory.ReservedCapacity += _memoryReservationStep;
                    SetPointers(_memory);
                    LinksHeader.SetReservedLinks(_header,
                    ↪   (Integer<TLink>)(_memory.ReservedCapacity / LinkSizeInBytes));
                }
                LinksHeader.SetAllocatedLinks(_header,
                ↪   Increment(LinksHeader.GetAllocatedLinks(_header)));
```

```csharp
                    _memory.UsedCapacity += LinkSizeInBytes;
                    freeLink = LinksHeader.GetAllocatedLinks(_header);
                }
                return freeLink;
            }

        public void Delete(TLink link)
        {
            if (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) < 0)
            {
                _unusedLinksListMethods.AttachAsFirst(link);
            }
            else if (_equalityComparer.Equals(link, LinksHeader.GetAllocatedLinks(_header)))
            {
                LinksHeader.SetAllocatedLinks(_header,
                    ↪ Decrement(LinksHeader.GetAllocatedLinks(_header)));
                _memory.UsedCapacity -= LinkSizeInBytes;
                // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
                    ↪ пока не дойдём до первой существующей связи
                // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
                while ((_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
                    ↪ Integer<TLink>.Zero) > 0) &&
                    ↪ IsUnusedLink(LinksHeader.GetAllocatedLinks(_header)))
                {
                    _unusedLinksListMethods.Detach(LinksHeader.GetAllocatedLinks(_header));
                    LinksHeader.SetAllocatedLinks(_header,
                        ↪ Decrement(LinksHeader.GetAllocatedLinks(_header)));
                    _memory.UsedCapacity -= LinkSizeInBytes;
                }
            }
        }

        /// <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(IDirectMemory memory)
        {
            if (memory == null)
            {
                _links = IntPtr.Zero;
                _header = _links;
                _unusedLinksListMethods = null;
                _targetsTreeMethods = null;
                _unusedLinksListMethods = null;
            }
            else
            {
                _links = memory.Pointer;
                _header = _links;
                _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
                _targetsTreeMethods = new LinksTargetsTreeMethods(this);
                _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool Exists(TLink link)
            => (_comparer.Compare(link, Constants.MinPossibleIndex) >= 0)
            && (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) <= 0)
            && !IsUnusedLink(link);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool IsUnusedLink(TLink link)
            => _equalityComparer.Equals(LinksHeader.GetFirstFreeLink(_header), link)
            || (_equalityComparer.Equals(Link.GetSizeAsSource(GetLinkUnsafe(link)),
            ↪ Constants.Null)
            && !_equalityComparer.Equals(Link.GetSource(GetLinkUnsafe(link)), Constants.Null));

        #region DisposableBase

        protected override bool AllowMultipleDisposeCalls => true;

        protected override void Dispose(bool manual, bool wasDisposed)
        {
```

```
581            if (!wasDisposed)
582            {
583                SetPointers(null);
584                _memory.DisposeIfPossible();
585            }
586        }
587
588        #endregion
589    }
590 }
```

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

```
1  using System;
2  using Platform.Unsafe;
3  using Platform.Collections.Methods.Lists;
4
5  namespace Platform.Data.Doublets.ResizableDirectMemory
6  {
7      partial class ResizableDirectMemoryLinks<TLink>
8      {
9          private class UnusedLinksListMethods : CircularDoublyLinkedListMethods<TLink>
10         {
11             private readonly IntPtr _links;
12             private readonly IntPtr _header;
13
14             public UnusedLinksListMethods(IntPtr links, IntPtr header)
15             {
16                 _links = links;
17                 _header = header;
18             }
19
20             protected override TLink GetFirst() => (_header +
              ↪   LinksHeader.FirstFreeLinkOffset).GetValue<TLink>();
21
22             protected override TLink GetLast() => (_header +
              ↪   LinksHeader.LastFreeLinkOffset).GetValue<TLink>();
23
24             protected override TLink GetPrevious(TLink element) =>
              ↪   (_links.GetElement(LinkSizeInBytes, element) +
              ↪   Link.SourceOffset).GetValue<TLink>();
25
26             protected override TLink GetNext(TLink element) =>
              ↪   (_links.GetElement(LinkSizeInBytes, element) +
              ↪   Link.TargetOffset).GetValue<TLink>();
27
28             protected override TLink GetSize() => (_header +
              ↪   LinksHeader.FreeLinksOffset).GetValue<TLink>();
29
30             protected override void SetFirst(TLink element) => (_header +
              ↪   LinksHeader.FirstFreeLinkOffset).SetValue(element);
31
32             protected override void SetLast(TLink element) => (_header +
              ↪   LinksHeader.LastFreeLinkOffset).SetValue(element);
33
34             protected override void SetPrevious(TLink element, TLink previous) =>
              ↪   (_links.GetElement(LinkSizeInBytes, element) +
              ↪   Link.SourceOffset).SetValue(previous);
35
36             protected override void SetNext(TLink element, TLink next) =>
              ↪   (_links.GetElement(LinkSizeInBytes, element) + Link.TargetOffset).SetValue(next);
37
38             protected override void SetSize(TLink size) => (_header +
              ↪   LinksHeader.FreeLinksOffset).SetValue(size);
39         }
40     }
41 }
```

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

```
1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Numbers;
6  using Platform.Unsafe;
7  using Platform.Collections.Methods.Trees;
8  using Platform.Data.Constants;
9
10 namespace Platform.Data.Doublets.ResizableDirectMemory
11 {
```

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

```csharp
                    {
                        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)
            {
            }

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

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

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

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

170          protected override TLink GetSize(TLink node)
171          {
172              var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
        ↪       Link.SizeAsSourceOffset).GetValue<TLink>();
173              return Bit.PartialRead(previousValue, 5, -5);
174          }

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

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

180          protected override void SetSize(TLink node, TLink size)
181          {
182              var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
        ↪       Link.SizeAsSourceOffset).GetValue<TLink>();
183              (Links.GetElement(LinkSizeInBytes, node) +
        ↪       Link.SizeAsSourceOffset).SetValue(Bit.PartialWrite(previousValue, size, 5,
        ↪       -5));
184          }

186          protected override bool GetLeftIsChild(TLink node)
187          {
188              var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
        ↪       Link.SizeAsSourceOffset).GetValue<TLink>();
189              return (Integer<TLink>)Bit.PartialRead(previousValue, 4, 1);
190          }

192          protected override void SetLeftIsChild(TLink node, bool value)
193          {
194              var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
        ↪       Link.SizeAsSourceOffset).GetValue<TLink>();
195              var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 4,
        ↪       1);
196              (Links.GetElement(LinkSizeInBytes, node) +
        ↪       Link.SizeAsSourceOffset).SetValue(modified);
197          }

199          protected override bool GetRightIsChild(TLink node)
200          {
201              var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
        ↪       Link.SizeAsSourceOffset).GetValue<TLink>();
202              return (Integer<TLink>)Bit.PartialRead(previousValue, 3, 1);
203          }

205          protected override void SetRightIsChild(TLink node, bool value)
206          {
207              var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
        ↪       Link.SizeAsSourceOffset).GetValue<TLink>();
208              var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 3,
        ↪       1);
209              (Links.GetElement(LinkSizeInBytes, node) +
        ↪       Link.SizeAsSourceOffset).SetValue(modified);
210          }

212          protected override sbyte GetBalance(TLink node)
213          {
214              var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
        ↪       Link.SizeAsSourceOffset).GetValue<TLink>();
215              var value = (ulong)(Integer<TLink>)Bit.PartialRead(previousValue, 0, 3);
216              var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
        ↪       124 : value & 3);
217              return unpackedValue;
```

```
218              }
219
220          protected override void SetBalance(TLink node, sbyte value)
221          {
222              var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                  ↪  Link.SizeAsSourceOffset).GetValue<TLink>();
223              var packagedValue = (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) | value &
                  ↪  3);
224              var modified = Bit.PartialWrite(previousValue, packagedValue, 0, 3);
225              (Links.GetElement(LinkSizeInBytes, node) +
                  ↪  Link.SizeAsSourceOffset).SetValue(modified);
226          }
227
228          protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
229          {
230              var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
                  ↪  Link.SourceOffset).GetValue<TLink>();
231              var secondSource = (Links.GetElement(LinkSizeInBytes, second) +
                  ↪  Link.SourceOffset).GetValue<TLink>();
232              return LessThan(firstSource, secondSource) ||
233                      (IsEquals(firstSource, secondSource) &&
                          LessThan((Links.GetElement(LinkSizeInBytes, first) +
                          ↪  Link.TargetOffset).GetValue<TLink>(),
                          ↪  (Links.GetElement(LinkSizeInBytes, second) +
                          ↪  Link.TargetOffset).GetValue<TLink>()));
234          }
235
236          protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
237          {
238              var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
                  ↪  Link.SourceOffset).GetValue<TLink>();
239              var secondSource = (Links.GetElement(LinkSizeInBytes, second) +
                  ↪  Link.SourceOffset).GetValue<TLink>();
240              return GreaterThan(firstSource, secondSource) ||
241                      (IsEquals(firstSource, secondSource) &&
                          GreaterThan((Links.GetElement(LinkSizeInBytes, first) +
                          ↪  Link.TargetOffset).GetValue<TLink>(),
                          ↪  (Links.GetElement(LinkSizeInBytes, second) +
                          ↪  Link.TargetOffset).GetValue<TLink>()));
242          }
243
244          protected override TLink GetTreeRoot() => (Header +
              ↪  LinksHeader.FirstAsSourceOffset).GetValue<TLink>();
245
246          protected override TLink GetBasePartValue(TLink link) =>
              ↪  (Links.GetElement(LinkSizeInBytes, link) + Link.SourceOffset).GetValue<TLink>();
247
248          /// <summary>
249          /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
              ↪  (концом)
250          /// по дереву (индексу) связей, отсортированному по Source, а затем по Target.
251          /// </summary>
252          /// <param name="source">Индекс связи, которая является началом на искомой
              ↪  связи.</param>
253          /// <param name="target">Индекс связи, которая является концом на искомой
              ↪  связи.</param>
254          /// <returns>Индекс искомой связи.</returns>
255          public TLink Search(TLink source, TLink target)
256          {
257              var root = GetTreeRoot();
258              while (!EqualToZero(root))
259              {
260                  var rootSource = (Links.GetElement(LinkSizeInBytes, root) +
                      ↪  Link.SourceOffset).GetValue<TLink>();
261                  var rootTarget = (Links.GetElement(LinkSizeInBytes, root) +
                      ↪  Link.TargetOffset).GetValue<TLink>();
262                  if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                      ↪  node.Key < root.Key
263                  {
264                      root = GetLeftOrDefault(root);
265                  }
266                  else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget))
                      ↪  // node.Key > root.Key
267                  {
268                      root = GetRightOrDefault(root);
269                  }
```

```csharp
                        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);
                }

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

```
326                          (Links.GetElement(LinkSizeInBytes, node) +
                             ↪  Link.SizeAsTargetOffset).SetValue(modified);
327                  }
328
329              protected override bool GetRightIsChild(TLink node)
330              {
331                  var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                     ↪  Link.SizeAsTargetOffset).GetValue<TLink>();
332                  return (Integer<TLink>)Bit.PartialRead(previousValue, 3, 1);
333              }
334
335              protected override void SetRightIsChild(TLink node, bool value)
336              {
337                  var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                     ↪  Link.SizeAsTargetOffset).GetValue<TLink>();
338                  var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 3,
                     ↪  1);
339                  (Links.GetElement(LinkSizeInBytes, node) +
                     ↪  Link.SizeAsTargetOffset).SetValue(modified);
340              }
341
342              protected override sbyte GetBalance(TLink node)
343              {
344                  var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                     ↪  Link.SizeAsTargetOffset).GetValue<TLink>();
345                  var value = (ulong)(Integer<TLink>)Bit.PartialRead(previousValue, 0, 3);
346                  var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
                     ↪  124 : value & 3);
347                  return unpackedValue;
348              }
349
350              protected override void SetBalance(TLink node, sbyte value)
351              {
352                  var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                     ↪  Link.SizeAsTargetOffset).GetValue<TLink>();
353                  var packagedValue = (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) | value &
                     ↪  3);
354                  var modified = Bit.PartialWrite(previousValue, packagedValue, 0, 3);
355                  (Links.GetElement(LinkSizeInBytes, node) +
                     ↪  Link.SizeAsTargetOffset).SetValue(modified);
356              }
357
358              protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
359              {
360                  var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
                     ↪  Link.TargetOffset).GetValue<TLink>();
361                  var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
                     ↪  Link.TargetOffset).GetValue<TLink>();
362                  return LessThan(firstTarget, secondTarget) ||
363                          (IsEquals(firstTarget, secondTarget) &&
                             ↪    LessThan((Links.GetElement(LinkSizeInBytes, first) +
                             ↪    Link.SourceOffset).GetValue<TLink>(),
                             ↪    (Links.GetElement(LinkSizeInBytes, second) +
                             ↪    Link.SourceOffset).GetValue<TLink>()));
364              }
365
366              protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
367              {
368                  var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
                     ↪  Link.TargetOffset).GetValue<TLink>();
369                  var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
                     ↪  Link.TargetOffset).GetValue<TLink>();
370                  return GreaterThan(firstTarget, secondTarget) ||
371                          (IsEquals(firstTarget, secondTarget) &&
                             ↪    GreaterThan((Links.GetElement(LinkSizeInBytes, first) +
                             ↪    Link.SourceOffset).GetValue<TLink>(),
                             ↪    (Links.GetElement(LinkSizeInBytes, second) +
                             ↪    Link.SourceOffset).GetValue<TLink>()));
372              }
373
374              protected override TLink GetTreeRoot() => (Header +
                 ↪  LinksHeader.FirstAsTargetOffset).GetValue<TLink>();
375
376              protected override TLink GetBasePartValue(TLink link) =>
                 ↪  (Links.GetElement(LinkSizeInBytes, link) + Link.TargetOffset).GetValue<TLink>();
377          }
378      }
```

```
379     }
```

./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  //#define ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
12
13  #pragma warning disable 0649
14  #pragma warning disable 169
15
16  // ReSharper disable BuiltInTypeReferenceStyle
17
18  namespace Platform.Data.Doublets.ResizableDirectMemory
19  {
20      using id = UInt64;
21
22      public unsafe partial class UInt64ResizableDirectMemoryLinks : DisposableBase, ILinks<id>
23      {
24          /// <summary>Возвращает размер одной связи в байтах.</summary>
25          /// <remarks>
26          /// Используется только во вне класса, не рекомедуется использовать внутри.
27          /// Так как во вне не обязательно будет доступен unsafe C#.
28          /// </remarks>
29          public static readonly int LinkSizeInBytes = sizeof(Link);
30
31          public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
32
33          private struct Link
34          {
35              public id Source;
36              public id Target;
37              public id LeftAsSource;
38              public id RightAsSource;
39              public id SizeAsSource;
40              public id LeftAsTarget;
41              public id RightAsTarget;
42              public id SizeAsTarget;
43          }
44
45          private struct LinksHeader
46          {
47              public id AllocatedLinks;
48              public id ReservedLinks;
49              public id FreeLinks;
50              public id FirstFreeLink;
51              public id FirstAsSource;
52              public id FirstAsTarget;
53              public id LastFreeLink;
54              public id Reserved8;
55          }
56
57          private readonly long _memoryReservationStep;
58
59          private readonly IResizableDirectMemory _memory;
60          private LinksHeader* _header;
61          private Link* _links;
62
63          private LinksTargetsTreeMethods _targetsTreeMethods;
64          private LinksSourcesTreeMethods _sourcesTreeMethods;
65
66          // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
67          //       нужно использовать не список а дерево, так как так можно быстрее проверить на
68          //       наличие связи внутри
69          private UnusedLinksListMethods _unusedLinksListMethods;
70
71          /// <summary>
72          /// Возвращает общее число связей находящихся в хранилище.
73          /// </summary>
74          private id Total => _header->AllocatedLinks - _header->FreeLinks;
75
76          // TODO: Дать возможность переопределять в конструкторе
77          public LinksCombinedConstants<id, id, int> Constants { get; }
78
79          public UInt64ResizableDirectMemoryLinks(string address) : this(address,
80              DefaultLinksSizeStep) { }
```

```csharp
        /// <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;
                    }
                    if (value == Constants.Any)
                    {
                        return 1;
                    }
                    var storedLinkValue = GetLinkUnsafe(index);
                    if (storedLinkValue->Source == value ||
                        storedLinkValue->Target == value)
                    {
```

```csharp
                            return 1;
                        }
                        return 0;
                    }
                }
            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 Total;
                    }
                    else if (source == Constants.Any)
                    {
                        return _targetsTreeMethods.CountUsages(target);
                    }
                    else if (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 link == Constants.Null ? 0UL : 1UL;
                    }
                }
                else
                {
                    if (!Exists(index))
                    {
                        return 0;
                    }
                    if (source == Constants.Any && target == Constants.Any)
                    {
                        return 1;
                    }
                    var storedLinkValue = GetLinkUnsafe(index);
                    if (source != Constants.Any && target != Constants.Any)
                    {
                        if (storedLinkValue->Source == source &&
                            storedLinkValue->Target == target)
                        {
                            return 1;
                        }
                        return 0;
                    }
                    var value = default(id);
                    if (source == Constants.Any)
                    {
                        value = target;
                    }
                    if (target == Constants.Any)
                    {
                        value = source;
                    }
                    if (storedLinkValue->Source == value ||
                        storedLinkValue->Target == value)
                    {
                        return 1;
                    }
                    return 0;
                }
            }
        throw new NotSupportedException("Другие размеры и способы ограничений не
        ↪ поддерживаются.");
    }

    [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++)
            {
```

```csharp
                    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);
                    }
                    else if (source == Constants.Any)
                    {
                        return _targetsTreeMethods.EachReference(target, handler);
                    }
                    else if (target == Constants.Any)
                    {
                        return _sourcesTreeMethods.EachReference(source, handler);
                    }
                    else //if(source != Any && target != Any)
                    {
```

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

```csharp
378                    _sourcesTreeMethods.Attach(new IntPtr(&_header->FirstAsSource), linkIndex);
379                }
380                if (link->Target != Constants.Null)
381                {
382                    _targetsTreeMethods.Attach(new IntPtr(&_header->FirstAsTarget), linkIndex);
383                }
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
385                leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
386                rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
387                if (leftTreeSize != rightTreeSize)
388                {
389                    throw new Exception("One of the trees is broken.");
390                }
#endif
392                return linkIndex;
393            }
394
395            [MethodImpl(MethodImplOptions.AggressiveInlining)]
396            private IList<id> GetLinkStruct(id linkIndex)
397            {
398                var link = GetLinkUnsafe(linkIndex);
399                return new UInt64Link(linkIndex, link->Source, link->Target);
400            }
401
402            [MethodImpl(MethodImplOptions.AggressiveInlining)]
403            private Link* GetLinkUnsafe(id linkIndex) => &_links[linkIndex];
404
405            /// <remarks>
406            /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
          ↪  пространство
407            /// </remarks>
408            public id Create()
409            {
410                var freeLink = _header->FirstFreeLink;
411                if (freeLink != Constants.Null)
412                {
413                    _unusedLinksListMethods.Detach(freeLink);
414                }
415                else
416                {
417                    if (_header->AllocatedLinks > Constants.MaxPossibleIndex)
418                    {
419                        throw new LinksLimitReachedException(Constants.MaxPossibleIndex);
420                    }
421                    if (_header->AllocatedLinks >= _header->ReservedLinks - 1)
422                    {
423                        _memory.ReservedCapacity += _memoryReservationStep;
424                        SetPointers(_memory);
425                        _header->ReservedLinks = (id)(_memory.ReservedCapacity / sizeof(Link));
426                    }
427                    _header->AllocatedLinks++;
428                    _memory.UsedCapacity += sizeof(Link);
429                    freeLink = _header->AllocatedLinks;
430                }
431                return freeLink;
432            }
433
434            public void Delete(id link)
435            {
436                if (link < _header->AllocatedLinks)
437                {
438                    _unusedLinksListMethods.AttachAsFirst(link);
439                }
440                else if (link == _header->AllocatedLinks)
441                {
442                    _header->AllocatedLinks--;
443                    _memory.UsedCapacity -= sizeof(Link);
444                    // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
                  ↪  пока не дойдём до первой существующей связи
445                    // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
446                    while (_header->AllocatedLinks > 0 && IsUnusedLink(_header->AllocatedLinks))
447                    {
448                        _unusedLinksListMethods.Detach(_header->AllocatedLinks);
449                        _header->AllocatedLinks--;
450                        _memory.UsedCapacity -= sizeof(Link);
451                    }
452                }
453            }
454
```

```csharp
        /// <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;

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;
                _header = header;
            }

            protected override ulong GetFirst() => _header->FirstFreeLink;

            protected override ulong GetLast() => _header->LastFreeLink;

            protected override ulong GetPrevious(ulong element) => _links[element].Source;

            protected override ulong GetNext(ulong element) => _links[element].Target;
```

```
26          protected override ulong GetSize() => _header->FreeLinks;

27
28          protected override void SetFirst(ulong element) => _header->FirstFreeLink = element;

29
30          protected override void SetLast(ulong element) => _header->LastFreeLink = element;

31
32          protected override void SetPrevious(ulong element, ulong previous) =>
            ↪  _links[element].Source = previous;
33
34          protected override void SetNext(ulong element, ulong next) => _links[element].Target
            ↪  = next;
35
36          protected override void SetSize(ulong size) => _header->FreeLinks = size;
37        }
38      }
39  }
```

./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   namespace Platform.Data.Doublets.ResizableDirectMemory
9   {
10      unsafe partial class UInt64ResizableDirectMemoryLinks
11      {
12          private abstract class LinksTreeMethodsBase :
            ↪  SizedAndThreadedAVLBalancedTreeMethods<ulong>
13          {
14              private readonly UInt64ResizableDirectMemoryLinks _memory;
15              private readonly LinksCombinedConstants<ulong, ulong, int> _constants;
16              protected readonly Link* Links;
17              protected readonly LinksHeader* Header;
18
19              protected LinksTreeMethodsBase(UInt64ResizableDirectMemoryLinks memory)
20              {
21                  Links = memory._links;
22                  Header = memory._header;
23                  _memory = memory;
24                  _constants = memory.Constants;
25              }
26
27              [MethodImpl(MethodImplOptions.AggressiveInlining)]
28              protected abstract ulong GetTreeRoot();
29
30              [MethodImpl(MethodImplOptions.AggressiveInlining)]
31              protected abstract ulong GetBasePartValue(ulong link);
32
33              public ulong this[ulong index]
34              {
35                  get
36                  {
37                      var root = GetTreeRoot();
38                      if (index >= GetSize(root))
39                      {
40                          return 0;
41                      }
42                      while (root != 0)
43                      {
44                          var left = GetLeftOrDefault(root);
45                          var leftSize = GetSizeOrZero(left);
46                          if (index < leftSize)
47                          {
48                              root = left;
49                              continue;
50                          }
51                          if (index == leftSize)
52                          {
53                              return root;
54                          }
55                          root = GetRightOrDefault(root);
56                          index -= leftSize + 1;
57                      }
58                      return 0; // TODO: Impossible situation exception (only if tree structure
                        ↪  broken)
59                  }
60              }
```

```csharp
            // TODO: Return indices range instead of references count
            public ulong CountUsages(ulong link)
            {
                var root = GetTreeRoot();
                var total = GetSize(root);
                var totalRightIgnore = 0UL;
                while (root != 0)
                {
                    var @base = GetBasePartValue(root);
                    if (@base <= link)
                    {
                        root = GetRightOrDefault(root);
                    }
                    else
                    {
                        totalRightIgnore += GetRightSize(root) + 1;
                        root = GetLeftOrDefault(root);
                    }
                }
                root = GetTreeRoot();
                var totalLeftIgnore = 0UL;
                while (root != 0)
                {
                    var @base = GetBasePartValue(root);
                    if (@base >= link)
                    {
                        root = GetLeftOrDefault(root);
                    }
                    else
                    {
                        totalLeftIgnore += GetLeftSize(root) + 1;
                        root = GetRightOrDefault(root);
                    }
                }
                return total - totalRightIgnore - totalLeftIgnore;
            }

            public ulong EachReference(ulong link, Func<IList<ulong>, ulong> handler)
            {
                var root = GetTreeRoot();
                if (root == 0)
                {
                    return _constants.Continue;
                }
                ulong first = 0, current = root;
                while (current != 0)
                {
                    var @base = GetBasePartValue(current);
                    if (@base >= link)
                    {
                        if (@base == link)
                        {
                            first = current;
                        }
                        current = GetLeftOrDefault(current);
                    }
                    else
                    {
                        current = GetRightOrDefault(current);
                    }
                }
                if (first != 0)
                {
                    current = first;
                    while (true)
                    {
                        if (handler(_memory.GetLinkStruct(current)) == _constants.Break)
                        {
                            return _constants.Break;
                        }
                        current = GetNext(current);
                        if (current == 0 || GetBasePartValue(current) != link)
                        {
                            break;
                        }
                    }
                }
                return _constants.Continue;
```

```csharp
            }

            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)
            {
                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;
            }
```

```csharp
215
216        protected override sbyte GetBalance(ulong node)
217        {
218            var previousValue = Links[node].SizeAsSource;
219            //var value = Math.PartialRead(previousValue, 0, 3);
220            var value = previousValue & 7;
221            var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
                 ↪ 124 : value & 3);
222            return unpackedValue;
223        }
224
225        protected override void SetBalance(ulong node, sbyte value)
226        {
227            var previousValue = Links[node].SizeAsSource;
228            var packagedValue = (ulong)(((((byte)value >> 5) & 4) | value & 3);
229            //var modified = Math.PartialWrite(previousValue, packagedValue, 0, 3);
230            var modified = (previousValue & 4294967288) | (packagedValue & 7);
231            Links[node].SizeAsSource = modified;
232        }
233
234        protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
235            => Links[first].Source < Links[second].Source ||
236               (Links[first].Source == Links[second].Source && Links[first].Target <
                 ↪ Links[second].Target);
237
238        protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
239            => Links[first].Source > Links[second].Source ||
240               (Links[first].Source == Links[second].Source && Links[first].Target >
                 ↪ Links[second].Target);
241
242        protected override ulong GetTreeRoot() => Header->FirstAsSource;
243
244        protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
245
246        /// <summary>
247        /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
             ↪ (концом)
248        /// по дереву (индексу) связей, отсортированному по Source, а затем по Target.
249        /// </summary>
250        /// <param name="source">Индекс связи, которая является началом на искомой
             ↪ связи.</param>
251        /// <param name="target">Индекс связи, которая является концом на искомой
             ↪ связи.</param>
252        /// <returns>Индекс искомой связи.</returns>
253        public ulong Search(ulong source, ulong target)
254        {
255            var root = Header->FirstAsSource;
256            while (root != 0)
257            {
258                var rootSource = Links[root].Source;
259                var rootTarget = Links[root].Target;
260                if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                     ↪ node.Key < root.Key
261                {
262                    root = GetLeftOrDefault(root);
263                }
264                else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget))
                     ↪ // node.Key > root.Key
265                {
266                    root = GetRightOrDefault(root);
267                }
268                else // node.Key == root.Key
269                {
270                    return root;
271                }
272            }
273            return 0;
274        }
275
276        [MethodImpl(MethodImplOptions.AggressiveInlining)]
277        private static bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
             ↪ ulong secondSource, ulong secondTarget)
278            => firstSource < secondSource || (firstSource == secondSource && firstTarget <
                 ↪ secondTarget);
279
280        [MethodImpl(MethodImplOptions.AggressiveInlining)]
281        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;

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

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

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

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

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

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

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

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

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

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

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

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

374        protected override ulong GetSize(ulong node)
375        {
376            var previousValue = Links[node].SizeAsTarget;
377            //return Math.PartialRead(previousValue, 5, -5);
378            return (previousValue & 4294967264) >> 5;
379        }

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

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

385        protected override void SetSize(ulong node, ulong size)
386        {
387            var previousValue = Links[node].SizeAsTarget;
388            //var modified = Math.PartialWrite(previousValue, size, 5, -5);
389            var modified = (previousValue & 31) | ((size & 134217727) << 5);
390            Links[node].SizeAsTarget = modified;
391        }

393        protected override bool GetLeftIsChild(ulong node)
394        {
395            var previousValue = Links[node].SizeAsTarget;
396            //return (Integer)Math.PartialRead(previousValue, 4, 1);
397            return (previousValue & 16) >> 4 == 1UL;
398            // TODO: Check if this is possible to use
399            //var nodeSize = GetSize(node);
400            //var left = GetLeftValue(node);
401            //var leftSize = GetSizeOrZero(left);
402            //return leftSize > 0 && nodeSize > leftSize;
403        }

405        protected override void SetLeftIsChild(ulong node, bool value)
406        {
407            var previousValue = Links[node].SizeAsTarget;
408            //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 4, 1);
409            var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
410            Links[node].SizeAsTarget = modified;
411        }

413        protected override bool GetRightIsChild(ulong node)
414        {
415            var previousValue = Links[node].SizeAsTarget;
416            //return (Integer)Math.PartialRead(previousValue, 3, 1);
417            return (previousValue & 8) >> 3 == 1UL;
418            // TODO: Check if this is possible to use
419            //var nodeSize = GetSize(node);
420            //var right = GetRightValue(node);
421            //var rightSize = GetSizeOrZero(right);
422            //return rightSize > 0 && nodeSize > rightSize;
```

```
423                    }
424
425            protected override void SetRightIsChild(ulong node, bool value)
426            {
427                var previousValue = Links[node].SizeAsTarget;
428                //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 3, 1);
429                var modified = (previousValue & 4294967287) | ((value ? 1UL : 0UL) << 3);
430                Links[node].SizeAsTarget = modified;
431            }
432
433            protected override sbyte GetBalance(ulong node)
434            {
435                var previousValue = Links[node].SizeAsTarget;
436                //var value = Math.PartialRead(previousValue, 0, 3);
437                var value = previousValue & 7;
438                var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
                     ↪  124 : value & 3);
439                return unpackedValue;
440            }
441
442            protected override void SetBalance(ulong node, sbyte value)
443            {
444                var previousValue = Links[node].SizeAsTarget;
445                var packagedValue = (ulong)((((byte)value >> 5) & 4) | value & 3);
446                //var modified = Math.PartialWrite(previousValue, packagedValue, 0, 3);
447                var modified = (previousValue & 4294967288) | (packagedValue & 7);
448                Links[node].SizeAsTarget = modified;
449            }
450
451            protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
452                => Links[first].Target < Links[second].Target ||
453                   (Links[first].Target == Links[second].Target && Links[first].Source <
                     ↪  Links[second].Source);
454
455            protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
456                => Links[first].Target > Links[second].Target ||
457                   (Links[first].Target == Links[second].Target && Links[first].Source >
                     ↪  Links[second].Source);
458
459            protected override ulong GetTreeRoot() => Header->FirstAsTarget;
460
461            protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
462
463            [MethodImpl(MethodImplOptions.AggressiveInlining)]
464            protected override void ClearNode(ulong node)
465            {
466                Links[node].LeftAsTarget = 0UL;
467                Links[node].RightAsTarget = 0UL;
468                Links[node].SizeAsTarget = 0UL;
469            }
470        }
471    }
472 }
```

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

```
1  using System.Collections.Generic;
2
3  namespace Platform.Data.Doublets.Sequences.Converters
4  {
5      public class BalancedVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
6      {
7          public BalancedVariantConverter(ILinks<TLink> links) : base(links) { }
8
9          public override TLink Convert(IList<TLink> sequence)
10         {
11             var length = sequence.Count;
12             if (length < 1)
13             {
14                 return default;
15             }
16             if (length == 1)
17             {
18                 return sequence[0];
19             }
20             // Make copy of next layer
21             if (length > 2)
22             {
23                 // TODO: Try to use stackalloc (which at the moment is not working with
                     ↪  generics) but will be possible with Sigil
```

```
24              var halvedSequence = new TLink[(length / 2) + (length % 2)];
25              HalveSequence(halvedSequence, sequence, length);
26              sequence = halvedSequence;
27              length = halvedSequence.Length;
28          }
29          // Keep creating layer after layer
30          while (length > 2)
31          {
32              HalveSequence(sequence, sequence, length);
33              length = (length / 2) + (length % 2);
34          }
35          return Links.GetOrCreate(sequence[0], sequence[1]);
36      }
37
38      private void HalveSequence(IList<TLink> destination, IList<TLink> source, int length)
39      {
40          var loopedLength = length - (length % 2);
41          for (var i = 0; i < loopedLength; i += 2)
42          {
43              destination[i / 2] = Links.GetOrCreate(source[i], source[i + 1]);
44          }
45          if (length > loopedLength)
46          {
47              destination[length / 2] = source[length - 1];
48          }
49      }
50  }
51 }
```

./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 namespace Platform.Data.Doublets.Sequences.Converters
12 {
13     /// <remarks>
14     /// TODO: Возможно будет лучше если алгоритм будет выполняться полностью изолированно от
          ↪ Links на этапе сжатия.
15     ///     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.
16     ///     Как только список/словарь пар был выявлен можно разом выполнить создание всех этих
          ↪ пар, а так же разом выполнить замену.
17     /// </remarks>
18     public class CompressingConverter<TLink> : LinksListToSequenceConverterBase<TLink>
19     {
20         private static readonly LinksCombinedConstants<bool, TLink, long> _constants =
              ↪ Default<LinksCombinedConstants<bool, TLink, long>>.Instance;
21         private static readonly EqualityComparer<TLink> _equalityComparer =
              ↪ EqualityComparer<TLink>.Default;
22         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
23
24         private readonly IConverter<IList<TLink>, TLink> _baseConverter;
25         private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
26         private readonly TLink _minFrequencyToCompress;
27         private readonly bool _doInitialFrequenciesIncrement;
28         private Doublet<TLink> _maxDoublet;
29         private LinkFrequency<TLink> _maxDoubletData;
30
31         private struct HalfDoublet
32         {
33             public TLink Element;
34             public LinkFrequency<TLink> DoubletData;
35
36             public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
37             {
38                 Element = element;
39                 DoubletData = doubletData;
40             }
41
42             public override string ToString() => $"{Element}: ({DoubletData})";
43         }
44
45         public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
              ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
```

```csharp
                 : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One, true)
        {
        }

        public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
        ↪  baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
        ↪  doInitialFrequenciesIncrement)
                 : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One,
                 ↪  doInitialFrequenciesIncrement)
        {
        }

        public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
        ↪  baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
        ↪  minFrequencyToCompress, bool doInitialFrequenciesIncrement)
                 : base(links)
        {
            _baseConverter = baseConverter;
            _doubletFrequenciesCache = doubletFrequenciesCache;
            if (_comparer.Compare(minFrequencyToCompress, Integer<TLink>.One) < 0)
            {
                minFrequencyToCompress = Integer<TLink>.One;
            }
            _minFrequencyToCompress = minFrequencyToCompress;
            _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
            ResetMaxDoublet();
        }

        public override TLink Convert(IList<TLink> source) =>
        ↪  _baseConverter.Convert(Compress(source));

        /// <remarks>
        /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding .
        /// Faster version (doublets' frequencies dictionary is not recreated).
        /// </remarks>
        private IList<TLink> Compress(IList<TLink> sequence)
        {
            if (sequence.IsNullOrEmpty())
            {
                return null;
            }
            if (sequence.Count == 1)
            {
                return sequence;
            }
            if (sequence.Count == 2)
            {
                return new[] { Links.GetOrCreate(sequence[0], sequence[1]) };
            }
            // TODO: arraypool with min size (to improve cache locality) or stackallow with Sigil
            var copy = new HalfDoublet[sequence.Count];
            Doublet<TLink> doublet = default;
            for (var i = 1; i < sequence.Count; i++)
            {
                doublet.Source = sequence[i - 1];
                doublet.Target = sequence[i];
                LinkFrequency<TLink> data;
                if (_doInitialFrequenciesIncrement)
                {
                    data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
                }
                else
                {
                    data = _doubletFrequenciesCache.GetFrequency(ref doublet);
                    if (data == null)
                    {
                        throw new NotSupportedException("If you ask not to increment
                        ↪  frequencies, it is expected that all frequencies for the sequence
                        ↪  are prepared.");
                    }
                }
                copy[i - 1].Element = sequence[i - 1];
                copy[i - 1].DoubletData = data;
                UpdateMaxDoublet(ref doublet, data);
            }
            copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
            copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
            if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
```

```csharp
116             {
117                 var newLength = ReplaceDoublets(copy);
118                 sequence = new TLink[newLength];
119                 for (int i = 0; i < newLength; i++)
120                 {
121                     sequence[i] = copy[i].Element;
122                 }
123             }
124             return sequence;
125         }
126
127         /// <remarks>
128         /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding
129         /// </remarks>
130         private int ReplaceDoublets(HalfDoublet[] copy)
131         {
132             var oldLength = copy.Length;
133             var newLength = copy.Length;
134             while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
135             {
136                 var maxDoubletSource = _maxDoublet.Source;
137                 var maxDoubletTarget = _maxDoublet.Target;
138                 if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
139                 {
140                     _maxDoubletData.Link = Links.GetOrCreate(maxDoubletSource, maxDoubletTarget);
141                 }
142                 var maxDoubletReplacementLink = _maxDoubletData.Link;
143                 oldLength--;
144                 var oldLengthMinusTwo = oldLength - 1;
145                 // Substitute all usages
146                 int w = 0, r = 0; // (r == read, w == write)
147                 for (; r < oldLength; r++)
148                 {
149                     if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
150                         _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
151                     {
152                         if (r > 0)
153                         {
154                             var previous = copy[w - 1].Element;
155                             copy[w - 1].DoubletData.DecrementFrequency();
156                             copy[w - 1].DoubletData =
157                                 _doubletFrequenciesCache.IncrementFrequency(previous,
158                                 maxDoubletReplacementLink);
159                         }
160                         if (r < oldLengthMinusTwo)
161                         {
162                             var next = copy[r + 2].Element;
163                             copy[r + 1].DoubletData.DecrementFrequency();
164                             copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(ma
165                                 xDoubletReplacementLink,
166                                 next);
167                         }
168                         copy[w++].Element = maxDoubletReplacementLink;
169                         r++;
170                         newLength--;
171                     }
172                     else
173                     {
174                         copy[w++] = copy[r];
175                     }
176                 }
177                 if (w < newLength)
178                 {
179                     copy[w] = copy[r];
180                 }
181                 oldLength = newLength;
182                 ResetMaxDoublet();
183                 UpdateMaxDoublet(copy, newLength);
184             }
185             return newLength;
186         }
187
188         [MethodImpl(MethodImplOptions.AggressiveInlining)]
189         private void ResetMaxDoublet()
190         {
191             _maxDoublet = new Doublet<TLink>();
192             _maxDoubletData = new LinkFrequency<TLink>();
193         }
```

```csharp
116             {
117                 var newLength = ReplaceDoublets(copy);
118                 sequence = new TLink[newLength];
119                 for (int i = 0; i < newLength; i++)
120                 {
121                     sequence[i] = copy[i].Element;
122                 }
123             }
124             return sequence;
125         }
126
127         /// <remarks>
128         /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding
129         /// </remarks>
130         private int ReplaceDoublets(HalfDoublet[] copy)
131         {
132             var oldLength = copy.Length;
133             var newLength = copy.Length;
134             while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
135             {
136                 var maxDoubletSource = _maxDoublet.Source;
137                 var maxDoubletTarget = _maxDoublet.Target;
138                 if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
139                 {
140                     _maxDoubletData.Link = Links.GetOrCreate(maxDoubletSource, maxDoubletTarget);
141                 }
142                 var maxDoubletReplacementLink = _maxDoubletData.Link;
143                 oldLength--;
144                 var oldLengthMinusTwo = oldLength - 1;
145                 // Substitute all usages
146                 int w = 0, r = 0; // (r == read, w == write)
147                 for (; r < oldLength; r++)
148                 {
149                     if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
                        ↪  _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
150                     {
151                         if (r > 0)
152                         {
153                             var previous = copy[w - 1].Element;
154                             copy[w - 1].DoubletData.DecrementFrequency();
155                             copy[w - 1].DoubletData =
                                ↪  _doubletFrequenciesCache.IncrementFrequency(previous,
                                ↪  maxDoubletReplacementLink);
156                         }
157                         if (r < oldLengthMinusTwo)
158                         {
159                             var next = copy[r + 2].Element;
160                             copy[r + 1].DoubletData.DecrementFrequency();
161                             copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(ma
                                ↪  xDoubletReplacementLink,
                                ↪  next);
162                         }
163                         copy[w++].Element = maxDoubletReplacementLink;
164                         r++;
165                         newLength--;
166                     }
167                     else
168                     {
169                         copy[w++] = copy[r];
170                     }
171                 }
172                 if (w < newLength)
173                 {
174                     copy[w] = copy[r];
175                 }
176                 oldLength = newLength;
177                 ResetMaxDoublet();
178                 UpdateMaxDoublet(copy, newLength);
179             }
180             return newLength;
181         }
182
183         [MethodImpl(MethodImplOptions.AggressiveInlining)]
184         private void ResetMaxDoublet()
185         {
186             _maxDoublet = new Doublet<TLink>();
187             _maxDoubletData = new LinkFrequency<TLink>();
188         }
189
```

```
190            [MethodImpl(MethodImplOptions.AggressiveInlining)]
191            private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
192            {
193                Doublet<TLink> doublet = default;
194                for (var i = 1; i < length; i++)
195                {
196                    doublet.Source = copy[i - 1].Element;
197                    doublet.Target = copy[i].Element;
198                    UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
199                }
200            }
201
202            [MethodImpl(MethodImplOptions.AggressiveInlining)]
203            private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
204            {
205                var frequency = data.Frequency;
206                var maxFrequency = _maxDoubletData.Frequency;
207                //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
          ↪    (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
          ↪    compression string data (and gives collisions quickly) */ _maxDoublet.Source +
          ↪    _maxDoublet.Target)))
208                if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
209                    (_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) */
210                {
211                    _maxDoublet = doublet;
212                    _maxDoubletData = data;
213                }
214            }
215        }
216    }
```

./Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs
```
1    using System.Collections.Generic;
2    using Platform.Interfaces;
3
4    namespace Platform.Data.Doublets.Sequences.Converters
5    {
6        public abstract class LinksListToSequenceConverterBase<TLink> : IConverter<IList<TLink>,
          ↪    TLink>
7        {
8            protected readonly ILinks<TLink> Links;
9            public LinksListToSequenceConverterBase(ILinks<TLink> links) => Links = links;
10            public abstract TLink Convert(IList<TLink> source);
11        }
12    }
```

./Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs
```
1    using System.Collections.Generic;
2    using System.Linq;
3    using Platform.Interfaces;
4
5    namespace Platform.Data.Doublets.Sequences.Converters
6    {
7        public class OptimalVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
8        {
9            private static readonly EqualityComparer<TLink> _equalityComparer =
          ↪    EqualityComparer<TLink>.Default;
10            private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
11
12            private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;
13
14            public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
          ↪    sequenceToItsLocalElementLevelsConverter) : base(links)
15                => _sequenceToItsLocalElementLevelsConverter =
                ↪    sequenceToItsLocalElementLevelsConverter;
16
17            public override TLink Convert(IList<TLink> sequence)
18            {
19                var length = sequence.Count;
20                if (length == 1)
21                {
22                    return sequence[0];
23                }
24                var links = Links;
```

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

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

101
102         private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
     ↪   => _comparer.Compare(previous, current) < 0 ? previous : current;

103     }
104 }
```

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

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

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

```
1   using Platform.Interfaces;
2
3   namespace Platform.Data.Doublets.Sequences.CreteriaMatchers
4   {
5       public class DefaultSequenceElementCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
     ↪   ICriterionMatcher<TLink>
6       {
7           public DefaultSequenceElementCriterionMatcher(ILinks<TLink> links) : base(links) { }
8           public bool IsMatched(TLink argument) => Links.IsPartialPoint(argument);
9       }
10  }
```

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

```
1   using System.Collections.Generic;
2   using Platform.Interfaces;
3
4   namespace Platform.Data.Doublets.Sequences.CreteriaMatchers
5   {
6       public class MarkedSequenceCriterionMatcher<TLink> : ICriterionMatcher<TLink>
7       {
8           private static readonly EqualityComparer<TLink> _equalityComparer =
     ↪   EqualityComparer<TLink>.Default;
9
10          private readonly ILinks<TLink> _links;
11          private readonly TLink _sequenceMarkerLink;
12
13          public MarkedSequenceCriterionMatcher(ILinks<TLink> links, TLink sequenceMarkerLink)
14          {
15              _links = links;
16              _sequenceMarkerLink = sequenceMarkerLink;
17          }
18
19          public bool IsMatched(TLink sequenceCandidate)
```

```
20            => _equalityComparer.Equals(_links.GetSource(sequenceCandidate), _sequenceMarkerLink)
21            || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarkerLink,
   ↪   sequenceCandidate), _links.Constants.Null);
22        }
23    }
```

./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   namespace Platform.Data.Doublets.Sequences
7   {
8       public class DefaultSequenceAppender<TLink> : LinksOperatorBase<TLink>,
         ↪   ISequenceAppender<TLink>
9       {
10          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪   EqualityComparer<TLink>.Default;
11
12          private readonly IStack<TLink> _stack;
13          private readonly ISequenceHeightProvider<TLink> _heightProvider;
14
15          public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
            ↪   ISequenceHeightProvider<TLink> heightProvider)
16              : base(links)
17          {
18              _stack = stack;
19              _heightProvider = heightProvider;
20          }
21
22          public TLink Append(TLink sequence, TLink appendant)
23          {
24              var cursor = sequence;
25              while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
26              {
27                  var source = Links.GetSource(cursor);
28                  var target = Links.GetTarget(cursor);
29                  if (_equalityComparer.Equals(_heightProvider.Get(source),
                    ↪   _heightProvider.Get(target)))
30                  {
31                      break;
32                  }
33                  else
34                  {
35                      _stack.Push(source);
36                      cursor = target;
37                  }
38              }
39              var left = cursor;
40              var right = appendant;
41              while (!_equalityComparer.Equals(cursor = _stack.Pop(), Links.Constants.Null))
42              {
43                  right = Links.GetOrCreate(left, right);
44                  left = cursor;
45              }
46              return Links.GetOrCreate(left, right);
47          }
48      }
49  }
```

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

```
1   using System.Collections.Generic;
2   using System.Linq;
3   using Platform.Interfaces;
4
5   namespace Platform.Data.Doublets.Sequences
6   {
7       public class DuplicateSegmentsCounter<TLink> : ICounter<int>
8       {
9           private readonly IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
            ↪   _duplicateFragmentsProvider;
10          public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLink>,
            ↪   IList<TLink>>>> duplicateFragmentsProvider) => _duplicateFragmentsProvider =
            ↪   duplicateFragmentsProvider;
11          public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
12      }
13  }
```

```csharp
using System;
using System.Linq;
using System.Collections.Generic;
using Platform.Interfaces;
using Platform.Collections;
using Platform.Collections.Lists;
using Platform.Collections.Segments;
using Platform.Collections.Segments.Walkers;
using Platform.Singletons;
using Platform.Numbers;
using Platform.Data.Sequences;
using Platform.Data.Doublets.Unicode;

namespace Platform.Data.Doublets.Sequences
{
    public class DuplicateSegmentsProvider<TLink> :
        DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
        IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
    {
        private readonly ILinks<TLink> _links;
        private readonly ISequences<TLink> _sequences;
        private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
        private BitString _visited;

        private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
            IList<TLink>>>
        {
            private readonly IListEqualityComparer<TLink> _listComparer;
            public ItemEquilityComparer() => _listComparer =
                Default<IListEqualityComparer<TLink>>.Instance;
            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);
            public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
                (_listComparer.GetHashCode(pair.Key),
                _listComparer.GetHashCode(pair.Value)).GetHashCode();
        }

        private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
        {
            private readonly IListComparer<TLink> _listComparer;

            public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;

            public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
                KeyValuePair<IList<TLink>, IList<TLink>> right)
            {
                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)
```

```csharp
69                        {
70                            WalkAll(sequenceElements);
71                        }
72                    }
73                    return _links.Constants.Continue;
74                });
75                var resultList = _groups.ToList();
76                var comparer = Default<ItemComparer>.Instance;
77                resultList.Sort(comparer);
#if DEBUG
79                foreach (var item in resultList)
80                {
81                    PrintDuplicates(item);
82                }
#endif
84                return resultList;
85            }
86
87        protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
    ↪   length) => new Segment<TLink>(elements, offset, length);
88
89        protected override void OnDublicateFound(Segment<TLink> segment)
90        {
91            var duplicates = CollectDuplicatesForSegment(segment);
92            if (duplicates.Count > 1)
93            {
94                _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
    ↪       duplicates));
95            }
96        }
97
98        private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
99        {
100            var duplicates = new List<TLink>();
101            var readAsElement = new HashSet<TLink>();
102            _sequences.Each(sequence =>
103            {
104                duplicates.Add(sequence);
105                readAsElement.Add(sequence);
106                return true; // Continue
107            }, segment);
108            if (duplicates.Any(x => _visited.Get((Integer<TLink>)x)))
109            {
110                return new List<TLink>();
111            }
112            foreach (var duplicate in duplicates)
113            {
114                var duplicateBitIndex = (long)(Integer<TLink>)duplicate;
115                _visited.Set(duplicateBitIndex);
116            }
117            if (_sequences is Sequences sequencesExperiments)
118            {
119                var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((H
    ↪       ashSet<ulong>)(object)readAsElement,
    ↪       (IList<ulong>)segment);
120                foreach (var partiallyMatchedSequence in partiallyMatched)
121                {
122                    TLink sequenceIndex = (Integer<TLink>)partiallyMatchedSequence;
123                    duplicates.Add(sequenceIndex);
124                }
125            }
126            duplicates.Sort();
127            return duplicates;
128        }
129
130        private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
131        {
132            if (!(_links is ILinks<ulong> ulongLinks))
133            {
134                return;
135            }
136            var duplicatesKey = duplicatesItem.Key;
137            var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
138            Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)})");
139            var duplicatesList = duplicatesItem.Value;
140            for (int i = 0; i < duplicatesList.Count; i++)
141            {
142                ulong sequenceIndex = (Integer<TLink>)duplicatesList[i];
```

```
143          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));
144          Console.WriteLine(formatedSequenceStructure);
145          var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
         ↪   ulongLinks);
146          Console.WriteLine(sequenceString);
147        }
148        Console.WriteLine();
149      }
150    }
151  }
```

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

```csharp
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using Platform.Interfaces;
5   using Platform.Numbers;
6
7   namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
8   {
9       /// <remarks>
10      /// Can be used to operate with many CompressingConverters (to keep global frequencies data
        ↪   between them).
11      /// TODO: Extract interface to implement frequencies storage inside Links storage
12      /// </remarks>
13      public class LinkFrequenciesCache<TLink> : LinksOperatorBase<TLink>
14      {
15          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪   EqualityComparer<TLink>.Default;
16          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
17
18          private readonly Dictionary<Doublet<TLink>, LinkFrequency<TLink>> _doubletsCache;
19          private readonly ICounter<TLink, TLink> _frequencyCounter;
20
21          public LinkFrequenciesCache(ILinks<TLink> links, ICounter<TLink, TLink> frequencyCounter)
22              : base(links)
23          {
24              _doubletsCache = new Dictionary<Doublet<TLink>, LinkFrequency<TLink>>(4096,
                ↪   DoubletComparer<TLink>.Default);
25              _frequencyCounter = frequencyCounter;
26          }
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
30          {
31              var doublet = new Doublet<TLink>(source, target);
32              return GetFrequency(ref doublet);
33          }
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
37          {
38              _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
39              return data;
40          }
41
42          public void IncrementFrequencies(IList<TLink> sequence)
43          {
44              for (var i = 1; i < sequence.Count; i++)
45              {
46                  IncrementFrequency(sequence[i - 1], sequence[i]);
47              }
48          }
49
50          [MethodImpl(MethodImplOptions.AggressiveInlining)]
51          public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
52          {
53              var doublet = new Doublet<TLink>(source, target);
54              return IncrementFrequency(ref doublet);
55          }
56
57          public void PrintFrequencies(IList<TLink> sequence)
58          {
59              for (var i = 1; i < sequence.Count; i++)
60              {
61                  PrintFrequency(sequence[i - 1], sequence[i]);
```

```csharp
                }
            }

        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))
                        || ((_comparer.Compare(count, frequency) > 0) &&
                           ↪  (_comparer.Compare(Arithmetic.Subtract(count, frequency),
                           ↪  Integer<TLink>.One) > 0)))
                    {
                        throw new InvalidOperationException("Frequencies validation failed.");
                    }
                }
                //else
                //{
                //    if (value.Frequency > 0)
                //    {
                //        var frequency = value.Frequency;
                //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
                //        var count = _countLinkFrequency(linkIndex);

                //        if ((frequency > count && frequency - count > 1) || (count > frequency
                ↪  && count - frequency > 1))
                //            throw new Exception("Frequencies validation failed.");
                //    }
                //}
            }
        }
    }
}
```

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

```csharp
using System.Runtime.CompilerServices;
using Platform.Numbers;

namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
{
    public class LinkFrequency<TLink>
    {
        public TLink Frequency { get; set; }
```

```csharp
 9          public TLink Link { get; set; }
10
11          public LinkFrequency(TLink frequency, TLink link)
12          {
13              Frequency = frequency;
14              Link = link;
15          }
16
17          public LinkFrequency() { }
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          public void IncrementFrequency() => Frequency = Arithmetic<TLink>.Increment(Frequency);
21
22          [MethodImpl(MethodImplOptions.AggressiveInlining)]
23          public void DecrementFrequency() => Frequency = Arithmetic<TLink>.Decrement(Frequency);
24
25          public override string ToString() => $"F: {Frequency}, L: {Link}";
26      }
27  }
```

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

```csharp
 1  using Platform.Interfaces;
 2
 3  namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
 4  {
 5      public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLink> :
        ↪  IConverter<Doublet<TLink>, TLink>
 6      {
 7          private readonly LinkFrequenciesCache<TLink> _cache;
 8          public
           ↪  FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink>
           ↪  cache) => _cache = cache;
 9          public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref source).Frequency;
10      }
11  }
```

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

```csharp
 1  using Platform.Interfaces;
 2
 3  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
 4  {
 5      public class MarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
        ↪  SequenceSymbolFrequencyOneOffCounter<TLink>
 6      {
 7          private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
 8
 9          public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
           ↪  ICriterionMatcher<TLink> markedSequenceMatcher, TLink sequenceLink, TLink symbol)
10              : base(links, sequenceLink, symbol)
11              => _markedSequenceMatcher = markedSequenceMatcher;
12
13          public override TLink Count()
14          {
15              if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
16              {
17                  return default;
18              }
19              return base.Count();
20          }
21      }
22  }
```

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

```csharp
 1  using System.Collections.Generic;
 2  using Platform.Interfaces;
 3  using Platform.Numbers;
 4  using Platform.Data.Sequences;
 5
 6  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
 7  {
 8      public class SequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
 9      {
10          private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
11          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
12
13          protected readonly ILinks<TLink> _links;
14          protected readonly TLink _sequenceLink;
15          protected readonly TLink _symbol;
16          protected TLink _total;
```

```
17
18          public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink sequenceLink,
            ↪   TLink symbol)
19          {
20              _links = links;
21              _sequenceLink = sequenceLink;
22              _symbol = symbol;
23              _total = default;
24          }
25
26          public virtual TLink Count()
27          {
28              if (_comparer.Compare(_total, default) > 0)
29              {
30                  return _total;
31              }
32              StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
                ↪   IsElement, VisitElement);
33              return _total;
34          }
35
36          private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
            ↪   _links.IsPartialPoint(x); // TODO: Use SequenceElementCreteriaMatcher instead of
            ↪   IsPartialPoint
37
38          private bool VisitElement(TLink element)
39          {
40              if (_equalityComparer.Equals(element, _symbol))
41              {
42                  _total = Arithmetic.Increment(_total);
43              }
44              return true;
45          }
46      }
47  }
```

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

```
1   using Platform.Interfaces;
2
3   namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
4   {
5       public class TotalMarkedSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
6       {
7           private readonly ILinks<TLink> _links;
8           private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
9
10          public TotalMarkedSequenceSymbolFrequencyCounter(ILinks<TLink> links,
            ↪   ICriterionMatcher<TLink> markedSequenceMatcher)
11          {
12              _links = links;
13              _markedSequenceMatcher = markedSequenceMatcher;
14          }
15
16          public TLink Count(TLink argument) => new
            ↪   TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
            ↪   _markedSequenceMatcher, argument).Count();
17      }
18  }
```

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

```
1   using Platform.Interfaces;
2   using Platform.Numbers;
3
4   namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
5   {
6       public class TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
        ↪   TotalSequenceSymbolFrequencyOneOffCounter<TLink>
7       {
8           private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
9
10          public TotalMarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
            ↪   ICriterionMatcher<TLink> markedSequenceMatcher, TLink symbol)
11              : base(links, symbol)
12              => _markedSequenceMatcher = markedSequenceMatcher;
13
14          protected override void CountSequenceSymbolFrequency(TLink link)
15          {
16              var symbolFrequencyCounter = new
                ↪   MarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
                ↪   _markedSequenceMatcher, link, _symbol);
```

```
17            _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
18        }
19     }
20  }
```

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

```
1   using Platform.Interfaces;
2
3   namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
4   {
5       public class TotalSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
6       {
7           private readonly ILinks<TLink> _links;
8           public TotalSequenceSymbolFrequencyCounter(ILinks<TLink> links) => _links = links;
9           public TLink Count(TLink symbol) => new
            ↪  TotalSequenceSymbolFrequencyOneOffCounter<TLink>(_links, symbol).Count();
10      }
11  }
```

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

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

```
59              {
60                  CountCore(doubletIndex);
61              }
62              return constants.Continue;
63          }
64      }
65  }
```

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

```
1   using System.Collections.Generic;
2   using Platform.Interfaces;
3
4   namespace Platform.Data.Doublets.Sequences.HeightProviders
5   {
6       public class CachedSequenceHeightProvider<TLink> : LinksOperatorBase<TLink>,
        ↪   ISequenceHeightProvider<TLink>
7       {
8           private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪   EqualityComparer<TLink>.Default;
9
10          private readonly TLink _heightPropertyMarker;
11          private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
12          private readonly IConverter<TLink> _addressToUnaryNumberConverter;
13          private readonly IConverter<TLink> _unaryNumberToAddressConverter;
14          private readonly IPropertiesOperator<TLink, TLink, TLink> _propertyOperator;
15
16          public CachedSequenceHeightProvider(
17              ILinks<TLink> links,
18              ISequenceHeightProvider<TLink> baseHeightProvider,
19              IConverter<TLink> addressToUnaryNumberConverter,
20              IConverter<TLink> unaryNumberToAddressConverter,
21              TLink heightPropertyMarker,
22              IPropertiesOperator<TLink, TLink, TLink> propertyOperator)
23              : base(links)
24          {
25              _heightPropertyMarker = heightPropertyMarker;
26              _baseHeightProvider = baseHeightProvider;
27              _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
28              _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
29              _propertyOperator = propertyOperator;
30          }
31
32          public TLink Get(TLink sequence)
33          {
34              TLink height;
35              var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
36              if (_equalityComparer.Equals(heightValue, default))
37              {
38                  height = _baseHeightProvider.Get(sequence);
39                  heightValue = _addressToUnaryNumberConverter.Convert(height);
40                  _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
41              }
42              else
43              {
44                  height = _unaryNumberToAddressConverter.Convert(heightValue);
45              }
46              return height;
47          }
48      }
49  }
```

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

```
1   using Platform.Interfaces;
2   using Platform.Numbers;
3
4   namespace Platform.Data.Doublets.Sequences.HeightProviders
5   {
6       public class DefaultSequenceRightHeightProvider<TLink> : LinksOperatorBase<TLink>,
        ↪   ISequenceHeightProvider<TLink>
7       {
8           private readonly ICriterionMatcher<TLink> _elementMatcher;
9
10          public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriterionMatcher<TLink>
            ↪   elementMatcher) : base(links) => _elementMatcher = elementMatcher;
11
12          public TLink Get(TLink sequence)
13          {
14              var height = default(TLink);
15              var pairOrElement = sequence;
16              while (!_elementMatcher.IsMatched(pairOrElement))
```

```
17                  {
18                      pairOrElement = Links.GetTarget(pairOrElement);
19                      height = Arithmetic.Increment(height);
20                  }
21                  return height;
22              }
23          }
24      }
```

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

```
1   using Platform.Interfaces;
2
3   namespace Platform.Data.Doublets.Sequences.HeightProviders
4   {
5       public interface ISequenceHeightProvider<TLink> : IProvider<TLink, TLink>
6       {
7       }
8   }
```

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

```
1   using System.Collections.Generic;
2   using Platform.Data.Doublets.Sequences.Frequencies.Cache;
3
4   namespace Platform.Data.Doublets.Sequences.Indexes
5   {
6       public class CachedFrequencyIncrementingSequenceIndex<TLink> : ISequenceIndex<TLink>
7       {
8           private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
9
10          private readonly LinkFrequenciesCache<TLink> _cache;
11
12          public CachedFrequencyIncrementingSequenceIndex(LinkFrequenciesCache<TLink> cache) =>
            ↪  _cache = cache;
13
14          public bool Add(IList<TLink> sequence)
15          {
16              var indexed = true;
17              var i = sequence.Count;
18              while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
                ↪  { }
19              for (; i >= 1; i--)
20              {
21                  _cache.IncrementFrequency(sequence[i - 1], sequence[i]);
22              }
23              return indexed;
24          }
25
26          private bool IsIndexedWithIncrement(TLink source, TLink target)
27          {
28              var frequency = _cache.GetFrequency(source, target);
29              if (frequency == null)
30              {
31                  return false;
32              }
33              var indexed = !_equalityComparer.Equals(frequency.Frequency, default);
34              if (indexed)
35              {
36                  _cache.IncrementFrequency(source, target);
37              }
38              return indexed;
39          }
40
41          public bool MightContain(IList<TLink> sequence)
42          {
43              var indexed = true;
44              var i = sequence.Count;
45              while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
46              return indexed;
47          }
48
49          private bool IsIndexed(TLink source, TLink target)
50          {
51              var frequency = _cache.GetFrequency(source, target);
52              if (frequency == null)
53              {
54                  return false;
55              }
56              return !_equalityComparer.Equals(frequency.Frequency, default);
57          }
```

```
58        }
59    }
```

./Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs
```
 1   using Platform.Interfaces;
 2   using System.Collections.Generic;
 3
 4   namespace Platform.Data.Doublets.Sequences.Indexes
 5   {
 6       public class FrequencyIncrementingSequenceIndex<TLink> : SequenceIndex<TLink>,
       ↪  ISequenceIndex<TLink>
 7       {
 8           private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
 9
10           private readonly IPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
11           private readonly IIncrementer<TLink> _frequencyIncrementer;
12
13           public FrequencyIncrementingSequenceIndex(ILinks<TLink> links, IPropertyOperator<TLink,
           ↪  TLink> frequencyPropertyOperator, IIncrementer<TLink> frequencyIncrementer)
14               : base(links)
15           {
16               _frequencyPropertyOperator = frequencyPropertyOperator;
17               _frequencyIncrementer = frequencyIncrementer;
18           }
19
20           public override bool Add(IList<TLink> sequence)
21           {
22               var indexed = true;
23               var i = sequence.Count;
24               while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
               ↪  { }
25               for (; i >= 1; i--)
26               {
27                   Increment(Links.GetOrCreate(sequence[i - 1], sequence[i]));
28               }
29               return indexed;
30           }
31
32           private bool IsIndexedWithIncrement(TLink source, TLink target)
33           {
34               var link = Links.SearchOrDefault(source, target);
35               var indexed = !_equalityComparer.Equals(link, default);
36               if (indexed)
37               {
38                   Increment(link);
39               }
40               return indexed;
41           }
42
43           private void Increment(TLink link)
44           {
45               var previousFrequency = _frequencyPropertyOperator.Get(link);
46               var frequency = _frequencyIncrementer.Increment(previousFrequency);
47               _frequencyPropertyOperator.Set(link, frequency);
48           }
49       }
50   }
```

./Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.cs
```
 1   using System.Collections.Generic;
 2
 3   namespace Platform.Data.Doublets.Sequences.Indexes
 4   {
 5       public interface ISequenceIndex<TLink>
 6       {
 7           /// <summary>
 8           /// Индексирует последовательность глобально, и возвращает значение,
 9           /// определяющие была ли запрошенная последовательность проиндексирована ранее.
10           /// </summary>
11           /// <param name="sequence">Последовательность для индексации.</param>
12           bool Add(IList<TLink> sequence);
13
14           bool MightContain(IList<TLink> sequence);
15       }
16   }
```

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

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

```
39                  {
40                      var indexed = true;
41                      var i = sequence.Count;
42                      while (--i >= 1 && (indexed =
    ↪  !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
    ↪  sequence[i]), default))) { }
43                      return indexed;
44                  });
45              }
46          }
47  }
```

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

```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  namespace Platform.Data.Doublets.Sequences
16  {
17      /// <summary>
18      /// Представляет коллекцию последовательностей связей.
19      /// </summary>
20      /// <remarks>
21      /// Обязательно реализовать атомарность каждого публичного метода.
22      ///
23      /// TODO:
24      ///
25      /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
26      /// через естественную группировку по unicode типам, все whitespace вместе, все символы
    ↪  вместе, все числа вместе и т.п.
27      /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
    ↪  графа)
28      ///
29      /// x*y - найти все связи между, в последовательностях любой формы, если не стоит
    ↪  ограничитель на то, что является последовательностью, а что нет,
30      /// то находятся любые структуры связей, которые содержат эти элементы именно в таком
    ↪  порядке.
31      ///
32      /// Рост последовательности слева и справа.
33      /// Поиск со звёздочкой.
34      /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
35      /// так же проблема может быть решена при реализации дистанционных триггеров.
36      /// Нужны ли уникальные указатели вообще?
37      /// Что если обращение к информации будет происходить через содержимое всегда?
38      ///
39      /// Писать тесты.
40      ///
41      ///
42      /// Можно убрать зависимость от конкретной реализации Links,
43      /// на зависимость от абстрактного элемента, который может быть представлен несколькими
    ↪  способами.
44      ///
45      /// Можно ли как-то сделать один общий интерфейс
46      ///
47      ///
48      /// Блокчейн и/или гит для распределённой записи транзакций.
49      ///
50      /// </remarks>
51      public partial class Sequences : ISequences<ulong> // IList<string>, IList<ulong[]> (после
    ↪  завершения реализации Sequences)
52      {
53          private static readonly LinksCombinedConstants<bool, ulong, long> _constants =
    ↪  Default<LinksCombinedConstants<bool, ulong, long>>.Instance;
54
55          /// <summary>Возвращает значение ulong, обозначающее любое количество связей.</summary>
56          public const ulong ZeroOrMany = ulong.MaxValue;
57
58          public SequencesOptions<ulong> Options;
59          public readonly SynchronizedLinks<ulong> Links;
60          public readonly ISynchronization Sync;
```

```csharp
        public Sequences(SynchronizedLinks<ulong> links)
            : this(links, new SequencesOptions<ulong>())
        {
        }

        public Sequences(SynchronizedLinks<ulong> links, SequencesOptions<ulong> options)
        {
            Links = links;
            Sync = links.SyncRoot;
            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;
            }
```

```csharp
140                 throw new NotImplementedException();
141             }
142
143         private ulong CountUsages(params ulong[] restrictions)
144         {
145             if (restrictions.Length == 0)
146             {
147                 return 0;
148             }
149             if (restrictions.Length == 1) // Первая связь это адрес
150             {
151                 if (restrictions[0] == _constants.Null)
152                 {
153                     return 0;
154                 }
155                 if (Options.UseSequenceMarker)
156                 {
157                     var elementsLink = GetSequenceElements(restrictions[0]);
158                     var sequenceLink = GetSequenceByElements(elementsLink);
159                     if (sequenceLink != _constants.Null)
160                     {
161                         return Links.Count(sequenceLink) + Links.Count(elementsLink) - 1;
162                     }
163                     return Links.Count(elementsLink);
164                 }
165                 return Links.Count(restrictions[0]);
166             }
167             throw new NotImplementedException();
168         }
169
170         #endregion
171
172         #region Create
173
174         public ulong Create(params ulong[] sequence)
175         {
176             return Sync.ExecuteWriteOperation(() =>
177             {
178                 if (sequence.IsNullOrEmpty())
179                 {
180                     return _constants.Null;
181                 }
182                 Links.EnsureEachLinkExists(sequence);
183                 return CreateCore(sequence);
184             });
185         }
186
187         private ulong CreateCore(params ulong[] sequence)
188         {
189             if (Options.UseIndex)
190             {
191                 Options.Index.Add(sequence);
192             }
193             var sequenceRoot = default(ulong);
194             if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
195             {
196                 var matches = Each(sequence);
197                 if (matches.Count > 0)
198                 {
199                     sequenceRoot = matches[0];
200                 }
201             }
202             else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
203             {
204                 return CompactCore(sequence);
205             }
206             if (sequenceRoot == default)
207             {
208                 sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
209             }
210             if (Options.UseSequenceMarker)
211             {
212                 Links.Unsync.CreateAndUpdate(Options.SequenceMarkerLink, sequenceRoot);
213             }
214             return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
215         }
216
217         #endregion
218
```

```csharp
219         #region Each
220
221         public List<ulong> Each(params ulong[] sequence)
222         {
223             var results = new List<ulong>();
224             Each(results.AddAndReturnTrue, sequence);
225             return results;
226         }
227
228         public bool Each(Func<ulong, bool> handler, IList<ulong> sequence)
229         {
230             return Sync.ExecuteReadOperation(() =>
231             {
232                 if (sequence.IsNullOrEmpty())
233                 {
234                     return true;
235                 }
236                 Links.EnsureEachLinkIsAnyOrExists(sequence);
237                 if (sequence.Count == 1)
238                 {
239                     var link = sequence[0];
240                     if (link == _constants.Any)
241                     {
242                         return Links.Unsync.Each(_constants.Any, _constants.Any, handler);
243                     }
244                     return handler(link);
245                 }
246                 if (sequence.Count == 2)
247                 {
248                     return Links.Unsync.Each(sequence[0], sequence[1], handler);
249                 }
250                 if (Options.UseIndex && !Options.Index.MightContain(sequence))
251                 {
252                     return false;
253                 }
254                 return EachCore(handler, sequence);
255             });
256         }
257
258         private bool EachCore(Func<ulong, bool> handler, IList<ulong> sequence)
259         {
260             var matcher = new Matcher(this, sequence, new HashSet<LinkIndex>(), handler);
261             // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
            ↪ Id.
262             Func<ulong, bool> innerHandler = Options.UseSequenceMarker ? (Func<ulong,
            ↪ bool>)matcher.HandleFullMatchedSequence : matcher.HandleFullMatched;
263             //if (sequence.Length >= 2)
264             if (!StepRight(innerHandler, sequence[0], sequence[1]))
265             {
266                 return false;
267             }
268             var last = sequence.Count - 2;
269             for (var i = 1; i < last; i++)
270             {
271                 if (!PartialStepRight(innerHandler, sequence[i], sequence[i + 1]))
272                 {
273                     return false;
274                 }
275             }
276             if (sequence.Count >= 3)
277             {
278                 if (!StepLeft(innerHandler, sequence[sequence.Count - 2],
                ↪ sequence[sequence.Count - 1]))
279                 {
280                     return false;
281                 }
282             }
283             return true;
284         }
285
286         private bool PartialStepRight(Func<ulong, bool> handler, ulong left, ulong right)
287         {
288             return Links.Unsync.Each(_constants.Any, left, doublet =>
289             {
290                 if (!StepRight(handler, doublet, right))
291                 {
292                     return false;
293                 }
```

```csharp
                if (left != doublet)
                {
                    return PartialStepRight(handler, doublet, right);
                }
                return true;
            });
        }

        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;
```

```csharp
                    if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
                        !sequence.EqualTo(newSequence))
                    {
                        bestVariant = CompactCore(newSequence);
                    }
                    else
                    {
                        bestVariant = CreateCore(newSequence);
                    }
                    // 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(() =>
```

```csharp
                {
                    // TODO: Check all options only ones before loop execution
                    foreach (var linkToDelete in Each(sequence))
                    {
                        DeleteOneCore(linkToDelete);
                    }
                });
        }

        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);
```

```csharp
        #endregion

        #region Garbage Collection

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

```csharp
595                    }
596                    return _filterPosition == _patternSequence.Count;
597                }
598
599            private bool FullMatchCore(LinkIndex element)
600            {
601                if (_filterPosition == _patternSequence.Count)
602                {
603                    _filterPosition = -2; // Длиннее чем нужно
604                    return false;
605                }
606                if (_patternSequence[_filterPosition] != _constants.Any
607                 && element != _patternSequence[_filterPosition])
608                {
609                    _filterPosition = -1;
610                    return false; // Начинается/Продолжается иначе
611                }
612                _filterPosition++;
613                return true;
614            }
615
616            public void AddFullMatchedToResults(ulong sequenceToMatch)
617            {
618                if (FullMatch(sequenceToMatch))
619                {
620                    _results.Add(sequenceToMatch);
621                }
622            }
623
624            public bool HandleFullMatched(ulong sequenceToMatch)
625            {
626                if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
627                {
628                    return _stopableHandler(sequenceToMatch);
629                }
630                return true;
631            }
632
633            public bool HandleFullMatchedSequence(ulong sequenceToMatch)
634            {
635                var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
636                if (sequence != _constants.Null && FullMatch(sequenceToMatch) &&
                    ↪  _results.Add(sequenceToMatch))
637                {
638                    return _stopableHandler(sequence);
639                }
640                return true;
641            }
642
643            /// <remarks>
644            /// TODO: Add support for LinksConstants.Any
645            /// </remarks>
646            public bool PartialMatch(LinkIndex sequenceToMatch)
647            {
648                _filterPosition = -1;
649                foreach (var part in Walk(sequenceToMatch))
650                {
651                    if (!PartialMatchCore(part))
652                    {
653                        break;
654                    }
655                }
656                return _filterPosition == _patternSequence.Count - 1;
657            }
658
659            private bool PartialMatchCore(LinkIndex element)
660            {
661                if (_filterPosition == (_patternSequence.Count - 1))
662                {
663                    return false; // Нашлось
664                }
665                if (_filterPosition >= 0)
666                {
667                    if (element == _patternSequence[_filterPosition + 1])
668                    {
669                        _filterPosition++;
670                    }
671                    else
672                    {
```

```csharp
                            _filterPosition = -1;
                        }
                    }
                    if (_filterPosition < 0)
                    {
                        if (element == _patternSequence[0])
                        {
                            _filterPosition = 0;
                        }
                    }
                    return true; // Ищем дальше
                }

                public void AddPartialMatchedToResults(ulong sequenceToMatch)
                {
                    if (PartialMatch(sequenceToMatch))
                    {
                        _results.Add(sequenceToMatch);
                    }
                }

                public bool HandlePartialMatched(ulong sequenceToMatch)
                {
                    if (PartialMatch(sequenceToMatch))
                    {
                        return _stopableHandler(sequenceToMatch);
                    }
                    return true;
                }

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

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

            #endregion
        }
}
```

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

```csharp
using System;
using LinkIndex = System.UInt64;
using System.Collections.Generic;
using Stack = System.Collections.Generic.Stack<ulong>;
using System.Linq;
using System.Text;
using Platform.Collections;
using Platform.Data.Exceptions;
using Platform.Data.Sequences;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Collections.Stacks;

namespace Platform.Data.Doublets.Sequences
{
    partial class Sequences
    {
        #region Create All Variants (Not Practical)

        /// <remarks>
```

```csharp
            /// Number of links that is needed to generate all variants for
            /// sequence of length N corresponds to https://oeis.org/A014143/list sequence.
            /// </remarks>
            public ulong[] CreateAllVariants2(ulong[] sequence)
            {
                return Sync.ExecuteWriteOperation(() =>
                {
                    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)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)Numbers.Math.Catalan(sequence.Length));
                    return CreateAllVariants1Core(sequence, results);
                });
            }
```

```csharp
96
97         private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
98         {
99             if (sequence.Length == 2)
100            {
101                var link = Links.Unsync.CreateAndUpdate(sequence[0], sequence[1]);
102                if (link == _constants.Null)
103                {
104                    throw new NotImplementedException("Creation cancellation is not
                     ↪ implemented.");
105                }
106                results.Add(link);
107                return results;
108            }
109            var innerSequenceLength = sequence.Length - 1;
110            var innerSequence = new ulong[innerSequenceLength];
111            for (var li = 0; li < innerSequenceLength; li++)
112            {
113                var link = Links.Unsync.CreateAndUpdate(sequence[li], sequence[li + 1]);
114                if (link == _constants.Null)
115                {
116                    throw new NotImplementedException("Creation cancellation is not
                     ↪ implemented.");
117                }
118                for (var isi = 0; isi < li; isi++)
119                {
120                    innerSequence[isi] = sequence[isi];
121                }
122                innerSequence[li] = link;
123                for (var isi = li + 1; isi < innerSequenceLength; isi++)
124                {
125                    innerSequence[isi] = sequence[isi + 1];
126                }
127                CreateAllVariants1Core(innerSequence, results);
128            }
129            return results;
130        }
131
132        #endregion
133
134        public HashSet<ulong> Each1(params ulong[] sequence)
135        {
136            var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
137            Each1(link =>
138            {
139                if (!visitedLinks.Contains(link))
140                {
141                    visitedLinks.Add(link); // изучить почему случаются повторы
142                }
143                return true;
144            }, sequence);
145            return visitedLinks;
146        }
147
148        private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
149        {
150            if (sequence.Length == 2)
151            {
152                Links.Unsync.Each(sequence[0], sequence[1], handler);
153            }
154            else
155            {
156                var innerSequenceLength = sequence.Length - 1;
157                for (var li = 0; li < innerSequenceLength; li++)
158                {
159                    var left = sequence[li];
160                    var right = sequence[li + 1];
161                    if (left == 0 && right == 0)
162                    {
163                        continue;
164                    }
165                    var linkIndex = li;
166                    ulong[] innerSequence = null;
167                    Links.Unsync.Each(left, right, doublet =>
168                    {
169                        if (innerSequence == null)
170                        {
171                            innerSequence = new ulong[innerSequenceLength];
```

```csharp
                                for (var isi = 0; isi < linkIndex; isi++)
                                {
                                    innerSequence[isi] = sequence[isi];
                                }
                                for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
                                {
                                    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;
                });
```

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

```csharp
            });
        }

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

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

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

        public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
        {
            return Sync.ExecuteReadOperation(() =>
            {
                var results = new List<ulong>();
                if (sequence.Length > 0)
                {
                    Links.EnsureEachLinkExists(sequence);
                    var firstElement = sequence[0];
                    if (sequence.Length == 1)
                    {
                        results.Add(firstElement);
                        return results;
                    }
                    if (sequence.Length == 2)
                    {
                        var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
                        if (doublet != _constants.Null)
                        {
                            results.Add(doublet);
                        }
                        return results;
                    }
                    var linksInSequence = new HashSet<ulong>(sequence);
                    void handler(ulong result)
                    {
                        var filterPosition = 0;
                        StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
                        ↪  Links.Unsync.GetTarget,
                            x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
                        ↪  x =>
                            {
                                if (filterPosition == sequence.Length)
                                {
                                    filterPosition = -2; // Длиннее чем нужно
                                    return false;
                                }
```

```csharp
                                    if (x != sequence[filterPosition])
                                    {
                                        filterPosition = -1;
                                        return false; // Начинается иначе
                                    }
                                    filterPosition++;

                                    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;
```

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

```csharp
                        elementToString(sb, element);
                        sb.Append('}');
                    }
                    else
                    {
                        elementToString(sb, element);
                    }
                    if (sb.Length < MaxSequenceFormatSize)
                    {
                        return true;
                    }
                    sb.Append(insertComma ? ", ..." : "...");
                    return false;
                });
            }
            sb.Append('}');
            return sb.ToString();
        }

        public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
        {
            return Sync.ExecuteReadOperation(() =>
            {
                if (sequence.Length > 0)
                {
                    Links.EnsureEachLinkExists(sequence);
                    var results = new HashSet<ulong>();
                    for (var i = 0; i < sequence.Length; i++)
                    {
                        AllUsagesCore(sequence[i], results);
                    }
                    var filteredResults = new List<ulong>();
                    var linksInSequence = new HashSet<ulong>(sequence);
                    foreach (var result in results)
                    {
                        var filterPosition = -1;
                        StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
                        ↪   Links.Unsync.GetTarget,
                            x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
                        ↪   x =>
                            {
                                if (filterPosition == (sequence.Length - 1))
                                {
                                    return false;
                                }
                                if (filterPosition >= 0)
                                {
                                    if (x == sequence[filterPosition + 1])
                                    {
                                        filterPosition++;
                                    }
                                    else
                                    {
                                        return false;
                                    }
                                }
                                if (filterPosition < 0)
                                {
                                    if (x == sequence[0])
                                    {
                                        filterPosition = 0;
                                    }
                                }
                                return true;
                            });
                        if (filterPosition == (sequence.Length - 1))
                        {
                            filteredResults.Add(result);
                        }
                    }
                    return filteredResults;
                }
                return new List<ulong>();
            });
        }

        public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
        {
            return Sync.ExecuteReadOperation(() =>
```

```csharp
620                     {
621                         if (sequence.Length > 0)
622                         {
623                             Links.EnsureEachLinkExists(sequence);
624                             var results = new HashSet<ulong>();
625                             for (var i = 0; i < sequence.Length; i++)
626                             {
627                                 AllUsagesCore(sequence[i], results);
628                             }
629                             var filteredResults = new HashSet<ulong>();
630                             var matcher = new Matcher(this, sequence, filteredResults, null);
631                             matcher.AddAllPartialMatchedToResults(results);
632                             return filteredResults;
633                         }
634                         return new HashSet<ulong>();
635                     });
636             }
637
638             public bool GetAllPartiallyMatchingSequences2(Func<ulong, bool> handler, params ulong[]
        ↪    sequence)
639             {
640                 return Sync.ExecuteReadOperation(() =>
641                 {
642                     if (sequence.Length > 0)
643                     {
644                         Links.EnsureEachLinkExists(sequence);
645
646                         var results = new HashSet<ulong>();
647                         var filteredResults = new HashSet<ulong>();
648                         var matcher = new Matcher(this, sequence, filteredResults, handler);
649                         for (var i = 0; i < sequence.Length; i++)
650                         {
651                             if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
652                             {
653                                 return false;
654                             }
655                         }
656                         return true;
657                     }
658                     return true;
659                 });
660             }
661
662             //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
663             //{
664             //    return Sync.ExecuteReadOperation(() =>
665             //    {
666             //        if (sequence.Length > 0)
667             //        {
668             //            _links.EnsureEachLinkIsAnyOrExists(sequence);
669
670             //            var firstResults = new HashSet<ulong>();
671             //            var lastResults = new HashSet<ulong>();
672
673             //            var first = sequence.First(x => x != LinksConstants.Any);
674             //            var last = sequence.Last(x => x != LinksConstants.Any);
675
676             //            AllUsagesCore(first, firstResults);
677             //            AllUsagesCore(last, lastResults);
678
679             //            firstResults.IntersectWith(lastResults);
680
681             //            //for (var i = 0; i < sequence.Length; i++)
682             //            //    AllUsagesCore(sequence[i], results);
683
684             //            var filteredResults = new HashSet<ulong>();
685             //            var matcher = new Matcher(this, sequence, filteredResults, null);
686             //            matcher.AddAllPartialMatchedToResults(firstResults);
687             //            return filteredResults;
688             //        }
689
690             //        return new HashSet<ulong>();
691             //    });
692             //}
693
694             public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
695             {
696                 return Sync.ExecuteReadOperation(() =>
697                 {
```

```
698                    if (sequence.Length > 0)
699                    {
700                        Links.EnsureEachLinkIsAnyOrExists(sequence);
701                        var firstResults = new HashSet<ulong>();
702                        var lastResults = new HashSet<ulong>();
703                        var first = sequence.First(x => x != _constants.Any);
704                        var last = sequence.Last(x => x != _constants.Any);
705                        AllUsagesCore(first, firstResults);
706                        AllUsagesCore(last, lastResults);
707                        firstResults.IntersectWith(lastResults);
708                        //for (var i = 0; i < sequence.Length; i++)
709                        //    AllUsagesCore(sequence[i], results);
710                        var filteredResults = new HashSet<ulong>();
711                        var matcher = new Matcher(this, sequence, filteredResults, null);
712                        matcher.AddAllPartialMatchedToResults(firstResults);
713                        return filteredResults;
714                    }
715                    return new HashSet<ulong>();
716                });
717        }
718
719        public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
    ↪    IList<ulong> sequence)
720        {
721            return Sync.ExecuteReadOperation(() =>
722            {
723                if (sequence.Count > 0)
724                {
725                    Links.EnsureEachLinkExists(sequence);
726                    var results = new HashSet<LinkIndex>();
727                    //var nextResults = new HashSet<ulong>();
728                    //for (var i = 0; i < sequence.Length; i++)
729                    //{
730                    //    AllUsagesCore(sequence[i], nextResults);
731                    //    if (results.IsNullOrEmpty())
732                    //    {
733                    //        results = nextResults;
734                    //        nextResults = new HashSet<ulong>();
735                    //    }
736                    //    else
737                    //    {
738                    //        results.IntersectWith(nextResults);
739                    //        nextResults.Clear();
740                    //    }
741                    //}
742                    var collector1 = new AllUsagesCollector1(Links.Unsync, results);
743                    collector1.Collect(Links.Unsync.GetLink(sequence[0]));
744                    var next = new HashSet<ulong>();
745                    for (var i = 1; i < sequence.Count; i++)
746                    {
747                        var collector = new AllUsagesCollector1(Links.Unsync, next);
748                        collector.Collect(Links.Unsync.GetLink(sequence[i]));
749
750                        results.IntersectWith(next);
751                        next.Clear();
752                    }
753                    var filteredResults = new HashSet<ulong>();
754                    var matcher = new Matcher(this, sequence, filteredResults, null,
        ↪    readAsElements);
755                    matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
        ↪    x)); // OrderBy is a Hack
756                    return filteredResults;
757                }
758                return new HashSet<ulong>();
759            });
760        }
761
762        // Does not work
763        public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
    ↪    params ulong[] sequence)
764        {
765            var visited = new HashSet<ulong>();
766            var results = new HashSet<ulong>();
767            var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
        ↪    true; }, readAsElements);
768            var last = sequence.Length - 1;
769            for (var i = 0; i < last; i++)
770            {
```

```csharp
                    PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
            }
            return results;
        }

        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)
                    //////{
```

```
844            //////    var usages = new HashSet<ulong>();
845            //////    for (int i = 0; i < sequence.Length; i++)
846            //////    {
847            //////        AllUsagesCore(sequence[i], usages);
848            //////    }
849            //////    //for (int i = 0; i < matches[0].Count; i++)
850            //////    //    AllUsagesCore(matches[0][i], usages);
851            //////    //usages.UnionWith(matches[0]);
852            //////    return usages.ToList();
853            //////}
854            var firstLinkUsages = new HashSet<ulong>();
855            AllUsagesCore(sequence[0], firstLinkUsages);
856            firstLinkUsages.Add(sequence[0]);
857            //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
         ↪  sequence[0] }; // or all sequences, containing this element?
858            //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
         ↪  1).ToList();
859            var results = new HashSet<ulong>();
860            foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
         ↪  firstLinkUsages, 1))
861            {
862                AllUsagesCore(match, results);
863            }
864            return results.ToList();
865        }
866        return new List<ulong>();
867    });
868    }
869
870    /// <remarks>
871    /// TODO: Может потроботаться ограничение на уровень глубины рекурсии
872    /// </remarks>
873    public HashSet<ulong> AllUsages(ulong link)
874    {
875        return Sync.ExecuteReadOperation(() =>
876        {
877            var usages = new HashSet<ulong>();
878            AllUsagesCore(link, usages);
879            return usages;
880        });
881    }
882
883    // При сборе всех использований (последовательностей) можно сохранять обратный путь к
     ↪  той связи с которой начинался поиск (STTTSSSTT),
884    // причём достаточно одного бита для хранения перехода влево или вправо
885    private void AllUsagesCore(ulong link, HashSet<ulong> usages)
886    {
887        bool handler(ulong doublet)
888        {
889            if (usages.Add(doublet))
890            {
891                AllUsagesCore(doublet, usages);
892            }
893            return true;
894        }
895        Links.Unsync.Each(link, _constants.Any, handler);
896        Links.Unsync.Each(_constants.Any, link, handler);
897    }
898
899    public HashSet<ulong> AllBottomUsages(ulong link)
900    {
901        return Sync.ExecuteReadOperation(() =>
902        {
903            var visits = new HashSet<ulong>();
904            var usages = new HashSet<ulong>();
905            AllBottomUsagesCore(link, visits, usages);
906            return usages;
907        });
908    }
909
910    private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
     ↪  usages)
911    {
912        bool handler(ulong doublet)
913        {
914            if (visits.Add(doublet))
915            {
916                AllBottomUsagesCore(doublet, visits, usages);
```

```csharp
                }
                return true;
            }
            if (Links.Unsync.Count(_constants.Any, link) == 0)
            {
                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)
```

```
 992                    {
 993                        if (_totals[link] == 0)
 994                        {
 995                            var total = 1UL;
 996                            _totals[link] = total;
 997                            var visitedChildren = new HashSet<ulong>();
 998                            bool linkCalculator(ulong child)
 999                            {
1000                                if (link != child && visitedChildren.Add(child))
1001                                {
1002                                    total += _totals[child] == 0 ? 1 : _totals[child];
1003                                }
1004                                return true;
1005                            }
1006                            _links.Unsync.Each(link, _constants.Any, linkCalculator);
1007                            _links.Unsync.Each(_constants.Any, link, linkCalculator);
1008                            _totals[link] = total;
1009                        }
1010                        return true;
1011                    }
1012            }

1013
1014        private class AllUsagesCalculator2
1015        {
1016            private readonly SynchronizedLinks<ulong> _links;
1017            private readonly ulong[] _totals;
1018
1019            public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1020            {
1021                _links = links;
1022                _totals = totals;
1023            }
1024
1025            public void Calculate() => _links.Each(_constants.Any, _constants.Any,
         ↪ CalculateCore);
1026
1027            private bool IsElement(ulong link)
1028            {
1029                //_linksInSequence.Contains(link) ||
1030                return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
             ↪ link;
1031            }
1032
1033            private bool CalculateCore(ulong link)
1034            {
1035                // TODO: Проработать защиту от зацикливания
1036                // Основано на SequenceWalker.WalkLeft
1037                Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1038                Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1039                Func<ulong, bool> isElement = IsElement;
1040                void visitLeaf(ulong parent)
1041                {
1042                    if (link != parent)
1043                    {
1044                        _totals[parent]++;
1045                    }
1046                }
1047                void visitNode(ulong parent)
1048                {
1049                    if (link != parent)
1050                    {
1051                        _totals[parent]++;
1052                    }
1053                }
1054                var stack = new Stack();
1055                var element = link;
1056                if (isElement(element))
1057                {
1058                    visitLeaf(element);
1059                }
1060                else
1061                {
1062                    while (true)
1063                    {
1064                        if (isElement(element))
1065                        {
1066                            if (stack.Count == 0)
1067                            {
1068                                break;
```

```csharp
                        }
                        element = stack.Pop();
                        var source = getSource(element);
                        var target = getTarget(element);
                        // 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)
```

```csharp
            {
                _links = links;
                _usages = usages;
            }

            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) // ?
            {
```

```csharp
                    return previousMatchings;
                }
            var secondLinkUsages = new HashSet<ulong>();
            AllUsagesCore(sequence[startAt], secondLinkUsages);
            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>();
```

```csharp
                    });
            }

            // Найти все возможные связи между указанным списком связей.
            // 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(() =>
```

```csharp
                {
                    var results = new BitString((long)Links.Unsync.Count() + 1); // new
                    ↪ BitArray((int)_links.Total + 1);
                    if (linksToConnect.Length > 0)
                    {
                        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 };
```

```csharp
                var simplifiedSequence = Simplify(sequence);
        }

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

        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)
                        {
```

```
1521                                CollectMatchingSequences(leftLink, leftBound, middleLinks,
                                ↪   elements[i], rightBound - 1, ref results);
1522                            }
1523                        }
1524                    }
1525                    else
1526                    {
1527                        for (var i = elements.Length - 1; i >= 0; i--)
1528                        {
1529                            var element = elements[i];
1530                            if (element != 0)
1531                            {
1532                                results.Add(element);
1533                            }
1534                        }
1535                    }
1536                }
1537            }
1538
1539            public ulong[] GetRightElements(ulong startLink, ulong rightLink)
1540            {
1541                var result = new ulong[5];
1542                TryStepRight(startLink, rightLink, result, 0);
1543                Links.Each(_constants.Any, startLink, couple =>
1544                {
1545                    if (couple != startLink)
1546                    {
1547                        if (TryStepRight(couple, rightLink, result, 2))
1548                        {
1549                            return false;
1550                        }
1551                    }
1552                    return true;
1553                });
1554                if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
1555                {
1556                    result[4] = startLink;
1557                }
1558                return result;
1559            }
1560
1561            public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
1562            {
1563                var added = 0;
1564                Links.Each(startLink, _constants.Any, couple =>
1565                {
1566                    if (couple != startLink)
1567                    {
1568                        var coupleTarget = Links.GetTarget(couple);
1569                        if (coupleTarget == rightLink)
1570                        {
1571                            result[offset] = couple;
1572                            if (++added == 2)
1573                            {
1574                                return false;
1575                            }
1576                        }
1577                        else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker
                            ↪   == Net.And &&
1578                        {
1579                            result[offset + 1] = couple;
1580                            if (++added == 2)
1581                            {
1582                                return false;
1583                            }
1584                        }
1585                    }
1586                    return true;
1587                });
1588                return added > 0;
1589            }
1590
1591            public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
1592            {
1593                var result = new ulong[5];
1594                TryStepLeft(startLink, leftLink, result, 0);
1595                Links.Each(startLink, _constants.Any, couple =>
1596                {
```

```csharp
                    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
```

```csharp
        public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
        ↪  HashSet<LinkIndex> results)
            : 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;
            _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
```

```csharp
                            {
                                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++)
        //    {
```

```csharp
1830    //          if (_patternSequence[_patternPosition] == Doublets.Links.Null)
1831    //              mininumGap++;
1832    //          else if (_patternSequence[_patternPosition] == ZeroOrMany)
1833    //              maximumGap = long.MaxValue;
1834    //          else
1835    //              break;
1836    //      }
1837
1838    //      if (maximumGap < mininumGap)
1839    //          maximumGap = mininumGap;
1840    //}
1841
1842    private bool PatternMatchCore(LinkIndex element)
1843    {
1844        if (_patternPosition >= _pattern.Count)
1845        {
1846            _patternPosition = -2;
1847            return false;
1848        }
1849        var currentPatternBlock = _pattern[_patternPosition];
1850        if (currentPatternBlock.Type == PatternBlockType.Gap)
1851        {
1852            //var currentMatchingBlockLength = (_sequencePosition -
1853            //    _lastMatchedBlockPosition);
1853            if (_sequencePosition < currentPatternBlock.Start)
1854            {
1855                _sequencePosition++;
1856                return true; // Двигаемся дальше
1857            }
1858            // Это последний блок
1859            if (_pattern.Count == _patternPosition + 1)
1860            {
1861                _patternPosition++;
1862                _sequencePosition = 0;
1863                return false; // Полное соответствие
1864            }
1865            else
1866            {
1867                if (_sequencePosition > currentPatternBlock.Stop)
1868                {
1869                    return false; // Соответствие невозможно
1870                }
1871                var nextPatternBlock = _pattern[_patternPosition + 1];
1872                if (_patternSequence[nextPatternBlock.Start] == element)
1873                {
1874                    if (nextPatternBlock.Start < nextPatternBlock.Stop)
1875                    {
1876                        _patternPosition++;
1877                        _sequencePosition = 1;
1878                    }
1879                    else
1880                    {
1881                        _patternPosition += 2;
1882                        _sequencePosition = 0;
1883                    }
1884                }
1885            }
1886        }
1887        else // currentPatternBlock.Type == PatternBlockType.Elements
1888        {
1889            var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1890            if (_patternSequence[patternElementPosition] != element)
1891            {
1892                return false; // Соответствие невозможно
1893            }
1894            if (patternElementPosition == currentPatternBlock.Stop)
1895            {
1896                _patternPosition++;
1897                _sequencePosition = 0;
1898            }
1899            else
1900            {
1901                _sequencePosition++;
1902            }
1903        }
1904        return true;
1905        //if (_patternSequence[_patternPosition] != element)
1906        //    return false;
1907        //else
```

```csharp
                    //{
                    //    _sequencePosition++;
                    //    _patternPosition++;
                    //    return true;
                    //}
                    ////////
                    //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;

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;
        }
    }
```

```
27    }

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

```
71                         }
72                         else
73                         {
74                             totalSequenceSymbolFrequencyCounter = new
                             ↪  TotalSequenceSymbolFrequencyCounter<TLink>(links);
75                         }
76                         var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
                         ↪  totalSequenceSymbolFrequencyCounter);
77                         var compressingConverter = new CompressingConverter<TLink>(links,
                         ↪  balancedVariantConverter, doubletFrequenciesCache);
78                         LinksToSequenceConverter = compressingConverter;
79                     }
80                 }
81                 else
82                 {
83                     if (LinksToSequenceConverter == null)
84                     {
85                         LinksToSequenceConverter = balancedVariantConverter;
86                     }
87                 }
88                 if (UseIndex && Index == null)
89                 {
90                     Index = new SequenceIndex<TLink>(links);
91                 }
92                 if (Walker == null)
93                 {
94                     Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
95                 }
96             }
97
98             public void ValidateOptions()
99             {
100                 if (UseGarbageCollection && !UseSequenceMarker)
101                 {
102                     throw new NotSupportedException("To use garbage collection UseSequenceMarker
                     ↪  option must be on.");
103                 }
104             }
105         }
106     }
```

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

```
1   using System.Collections.Generic;
2
3   namespace Platform.Data.Doublets.Sequences.Walkers
4   {
5       public interface ISequenceWalker<TLink>
6       {
7           IEnumerable<TLink> Walk(TLink sequence);
8       }
9   }
```

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

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Collections.Stacks;
4
5   namespace Platform.Data.Doublets.Sequences.Walkers
6   {
7       public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
8       {
9           public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links, stack)
            ↪  { }
10
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          protected override TLink GetNextElementAfterPop(TLink element) =>
            ↪  Links.GetSource(element);
13
14          [MethodImpl(MethodImplOptions.AggressiveInlining)]
15          protected override TLink GetNextElementAfterPush(TLink element) =>
            ↪  Links.GetTarget(element);
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          protected override IEnumerable<TLink> WalkContents(TLink element)
19          {
20              var parts = Links.GetLink(element);
21              var start = Links.Constants.IndexPart + 1;
22              for (var i = parts.Count - 1; i >= start; i--)
```

```csharp
23              {
24                  var part = parts[i];
25                  if (IsElement(part))
26                  {
27                      yield return part;
28                  }
29              }
30          }
31      }
32  }
```

./Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs
```csharp
 1  using System.Collections.Generic;
 2  using System.Runtime.CompilerServices;
 3
 4  //#define USEARRAYPOOL
 5  #if USEARRAYPOOL
 6  using Platform.Collections;
 7  #endif
 8
 9  namespace Platform.Data.Doublets.Sequences.Walkers
10  {
11      public class LeveledSequenceWalker<TLink> : LinksOperatorBase<TLink>, ISequenceWalker<TLink>
12      {
13          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
14
15          public LeveledSequenceWalker(ILinks<TLink> links) : base(links) { }
16
17          public IEnumerable<TLink> Walk(TLink sequence) => ToArray(sequence);
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          protected virtual bool IsElement(TLink elementLink) => Links.IsPartialPoint(elementLink);
21
22          public TLink[] ToArray(TLink sequence)
23          {
24              var length = 1;
25              var array = new TLink[length];
26              array[0] = sequence;
27              if (IsElement(sequence))
28              {
29                  return array;
30              }
31              bool hasElements;
32              do
33              {
34                  length *= 2;
35  #if USEARRAYPOOL
36                  var nextArray = ArrayPool.Allocate<ulong>(length);
37  #else
38                  var nextArray = new TLink[length];
39  #endif
40                  hasElements = false;
41                  for (var i = 0; i < array.Length; i++)
42                  {
43                      var candidate = array[i];
44                      if (_equalityComparer.Equals(array[i], default))
45                      {
46                          continue;
47                      }
48                      var doubletOffset = i * 2;
49                      if (IsElement(candidate))
50                      {
51                          nextArray[doubletOffset] = candidate;
52                      }
53                      else
54                      {
55                          var link = Links.GetLink(candidate);
56                          var linkSource = Links.GetSource(link);
57                          var linkTarget = Links.GetTarget(link);
58                          nextArray[doubletOffset] = linkSource;
59                          nextArray[doubletOffset + 1] = linkTarget;
60                          if (!hasElements)
61                          {
62                              hasElements = !(IsElement(linkSource) && IsElement(linkTarget));
63                          }
64                      }
65                  }
66  #if USEARRAYPOOL
67                  if (array.Length > 1)
```

```
68                 {
69                     ArrayPool.Free(array);
70                 }
71  #endif
72                 array = nextArray;
73             }
74             while (hasElements);
75             var filledElementsCount = CountFilledElements(array);
76             if (filledElementsCount == array.Length)
77             {
78                 return array;
79             }
80             else
81             {
82                 return CopyFilledElements(array, filledElementsCount);
83             }
84         }
85
86         [MethodImpl(MethodImplOptions.AggressiveInlining)]
87         private static TLink[] CopyFilledElements(TLink[] array, int filledElementsCount)
88         {
89             var finalArray = new TLink[filledElementsCount];
90             for (int i = 0, j = 0; i < array.Length; i++)
91             {
92                 if (!_equalityComparer.Equals(array[i], default))
93                 {
94                     finalArray[j] = array[i];
95                     j++;
96                 }
97             }
98  #if USEARRAYPOOL
99                 ArrayPool.Free(array);
100 #endif
101             return finalArray;
102         }
103
104         [MethodImpl(MethodImplOptions.AggressiveInlining)]
105         private static int CountFilledElements(TLink[] array)
106         {
107             var count = 0;
108             for (var i = 0; i < array.Length; i++)
109             {
110                 if (!_equalityComparer.Equals(array[i], default))
111                 {
112                     count++;
113                 }
114             }
115             return count;
116         }
117     }
118 }
```

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

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Collections.Stacks;
4
5   namespace Platform.Data.Doublets.Sequences.Walkers
6   {
7       public class RightSequenceWalker<TLink> : SequenceWalkerBase<TLink>
8       {
9           public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links,
            ↪  stack) { }
10
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          protected override TLink GetNextElementAfterPop(TLink element) =>
            ↪  Links.GetTarget(element);
13
14          [MethodImpl(MethodImplOptions.AggressiveInlining)]
15          protected override TLink GetNextElementAfterPush(TLink element) =>
            ↪  Links.GetSource(element);
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          protected override IEnumerable<TLink> WalkContents(TLink element)
19          {
20              var parts = Links.GetLink(element);
21              for (var i = Links.Constants.IndexPart + 1; i < parts.Count; i++)
22              {
23                  var part = parts[i];
```

```
24              if (IsElement(part))
25              {
26                  yield return part;
27              }
28          }
29      }
30  }
31 }
```

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

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

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

```
1  using System.Collections.Generic;
2  using Platform.Collections.Stacks;
3
4  namespace Platform.Data.Doublets.Stacks
5  {
6      public class Stack<TLink> : IStack<TLink>
```

```
 7      {
 8          private static readonly EqualityComparer<TLink> _equalityComparer =
                ↪  EqualityComparer<TLink>.Default;
 9
10          private readonly ILinks<TLink> _links;
11          private readonly TLink _stack;
12
13          public bool IsEmpty => _equalityComparer.Equals(Peek(), _stack);
14
15          public Stack(ILinks<TLink> links, TLink stack)
16          {
17              _links = links;
18              _stack = stack;
19          }
20
21          private TLink GetStackMarker() => _links.GetSource(_stack);
22
23          private TLink GetTop() => _links.GetTarget(_stack);
24
25          public TLink Peek() => _links.GetTarget(GetTop());
26
27          public TLink Pop()
28          {
29              var element = Peek();
30              if (!_equalityComparer.Equals(element, _stack))
31              {
32                  var top = GetTop();
33                  var previousTop = _links.GetSource(top);
34                  _links.Update(_stack, GetStackMarker(), previousTop);
35                  _links.Delete(top);
36              }
37              return element;
38          }
39
40          public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
                ↪  _links.GetOrCreate(GetTop(), element));
41      }
42  }
```

./Platform.Data.Doublets/Stacks/StackExtensions.cs
```
 1  namespace Platform.Data.Doublets.Stacks
 2  {
 3      public static class StackExtensions
 4      {
 5          public static TLink CreateStack<TLink>(this ILinks<TLink> links, TLink stackMarker)
 6          {
 7              var stackPoint = links.CreatePoint();
 8              var stack = links.Update(stackPoint, stackMarker, stackPoint);
 9              return stack;
10          }
11      }
12  }
```

./Platform.Data.Doublets/SynchronizedLinks.cs
```
 1  using System;
 2  using System.Collections.Generic;
 3  using Platform.Data.Constants;
 4  using Platform.Data.Doublets;
 5  using Platform.Threading.Synchronization;
 6
 7  namespace Platform.Data.Doublets
 8  {
 9      /// <remarks>
10      /// TODO: Autogeneration of synchronized wrapper (decorator).
11      /// TODO: Try to unfold code of each method using IL generation for performance improvements.
12      /// TODO: Or even to unfold multiple layers of implementations.
13      /// </remarks>
14      public class SynchronizedLinks<T> : ISynchronizedLinks<T>
15      {
16          public LinksCombinedConstants<T, T, int> Constants { get; }
17          public ISynchronization SyncRoot { get; }
18          public ILinks<T> Sync { get; }
19          public ILinks<T> Unsync { get; }
20
21          public SynchronizedLinks(ILinks<T> links) : this(new ReaderWriterLockSynchronization(),
                ↪  links) { }
22
23          public SynchronizedLinks(ISynchronization synchronization, ILinks<T> links)
24          {
```

```
25              SyncRoot = synchronization;
26              Sync = this;
27              Unsync = links;
28              Constants = links.Constants;
29          }
30
31          public T Count(IList<T> restriction) => SyncRoot.ExecuteReadOperation(restriction,
            ↪   Unsync.Count);
32          public T Each(Func<IList<T>, T> handler, IList<T> restrictions) =>
            ↪   SyncRoot.ExecuteReadOperation(handler, restrictions, (handler1, restrictions1) =>
            ↪   Unsync.Each(handler1, restrictions1));
33          public T Create() => SyncRoot.ExecuteWriteOperation(Unsync.Create);
34          public T Update(IList<T> restrictions) => SyncRoot.ExecuteWriteOperation(restrictions,
            ↪   Unsync.Update);
35          public void Delete(T link) => SyncRoot.ExecuteWriteOperation(link, Unsync.Delete);
36
37          //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
            ↪   IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
38          //{
39          //    if (restriction != null && substitution != null &&
            ↪   !substitution.EqualTo(restriction))
40          //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
            ↪   substitution, substitutedHandler, Unsync.Trigger);
41
42          //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
            ↪   substitutedHandler, Unsync.Trigger);
43          //}
44      }
45  }
```

./Platform.Data.Doublets/UInt64Link.cs

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

```csharp
        {
            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;
                }
                throw new NotSupportedException(); // Impossible path due to
                  Ensure.ArgumentInRange
            }
            set => throw new NotSupportedException();
        }

        public int Count => Length;

        public bool IsReadOnly => true;

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

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

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

./Platform.Data.Doublets/UInt64LinkExtensions.cs

```csharp
1  namespace Platform.Data.Doublets
2  {
3      public static class UInt64LinkExtensions
4      {
5          public static bool IsFullPoint(this UInt64Link link) => Point<ulong>.IsFullPoint(link);
6          public static bool IsPartialPoint(this UInt64Link link) =>
              ↪  Point<ulong>.IsPartialPoint(link);
7      }
8  }
```

./Platform.Data.Doublets/UInt64LinksExtensions.cs

```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  namespace Platform.Data.Doublets
10  {
11      public static class UInt64LinksExtensions
12      {
13          public static readonly LinksCombinedConstants<bool, ulong, int> Constants =
              ↪  Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
14
15          public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
16
17          public static void EnsureEachLinkExists(this ILinks<ulong> links, IList<ulong> sequence)
18          {
```

```csharp
                if (sequence == null)
                {
                    return;
                }
                for (var i = 0; i < sequence.Count; i++)
                {
                    if (!links.Exists(sequence[i]))
                    {
                        throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                        ↪  $"sequence[{i}]");
                    }
                }
            }

            public static void EnsureEachLinkIsAnyOrExists(this ILinks<ulong> links, IList<ulong>
            ↪  sequence)
            {
                if (sequence == null)
                {
                    return;
                }
                for (var i = 0; i < sequence.Count; i++)
                {
                    if (sequence[i] != Constants.Any && !links.Exists(sequence[i]))
                    {
                        throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                        ↪  $"sequence[{i}]");
                    }
                }
            }

            public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
            {
                if (sequence == null)
                {
                    return false;
                }
                var constants = links.Constants;
                for (var i = 0; i < sequence.Length; i++)
                {
                    if (sequence[i] == constants.Any)
                    {
                        return true;
                    }
                }
                return false;
            }

            public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
            ↪  Func<UInt64Link, bool> isElement, bool renderIndex = false, bool renderDebug = false)
            {
                var sb = new StringBuilder();
                var visited = new HashSet<ulong>();
                links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
                ↪  innerSb.Append(link.Index), renderIndex, renderDebug);
                return sb.ToString();
            }

            public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
            ↪  Func<UInt64Link, bool> isElement, Action<StringBuilder, UInt64Link> appendElement,
            ↪  bool renderIndex = false, bool renderDebug = false)
            {
                var sb = new StringBuilder();
                var visited = new HashSet<ulong>();
                links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
                ↪  renderDebug);
                return sb.ToString();
            }

            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)
            {
                if (sb == null)
                {
                    throw new ArgumentNullException(nameof(sb));
                }
```

```
 86              if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
   ↪    Constants.Itself)
 87              {
 88                  return;
 89              }
 90              if (links.Exists(linkIndex))
 91              {
 92                  if (visited.Add(linkIndex))
 93                  {
 94                      sb.Append('(');
 95                      var link = new UInt64Link(links.GetLink(linkIndex));
 96                      if (renderIndex)
 97                      {
 98                          sb.Append(link.Index);
 99                          sb.Append(':');
100                      }
101                      if (link.Source == link.Index)
102                      {
103                          sb.Append(link.Index);
104                      }
105                      else
106                      {
107                          var source = new UInt64Link(links.GetLink(link.Source));
108                          if (isElement(source))
109                          {
110                              appendElement(sb, source);
111                          }
112                          else
113                          {
114                              links.AppendStructure(sb, visited, source.Index, isElement,
   ↪        appendElement, renderIndex);
115                          }
116                      }
117                      sb.Append(' ');
118                      if (link.Target == link.Index)
119                      {
120                          sb.Append(link.Index);
121                      }
122                      else
123                      {
124                          var target = new UInt64Link(links.GetLink(link.Target));
125                          if (isElement(target))
126                          {
127                              appendElement(sb, target);
128                          }
129                          else
130                          {
131                              links.AppendStructure(sb, visited, target.Index, isElement,
   ↪        appendElement, renderIndex);
132                          }
133                      }
134                      sb.Append(')');
135                  }
136                  else
137                  {
138                      if (renderDebug)
139                      {
140                          sb.Append('*');
141                      }
142                      sb.Append(linkIndex);
143                  }
144              }
145              else
146              {
147                  if (renderDebug)
148                  {
149                      sb.Append('~');
150                  }
151                  sb.Append(linkIndex);
152              }
153          }
154      }
155  }
```

./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs
```
1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
```

```csharp
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;

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 & TransactionIdCombined;
        ///         }
        ///     }
        ///
        ///     public UniqueTimestamp Timestamp
        ///     {
        ///         get
        ///         {
        ///             return (UniqueTimestamp)mask & TransactionIdCombined;
        ///         }
        ///     }
        ///
        ///     public TransactionItemType Type
        ///     {
        ///         get
        ///         {
        ///             // Использовать по одному биту из TransactionId и Timestamp,
        ///             // для значения в 2 бита, которое представляет тип операции
        ///             throw new NotImplementedException();
        ///         }
        ///     }
        /// }
        ///
        /// private struct Transition
        /// {
        ///     public TransitionHeader Header;
        ///     public Link Source;
        ///     public Link Linker;
        ///     public Link Target;
        /// }
        ///
        /// </remarks>
```

```csharp
82      public struct Transition
83      {
84          public static readonly long Size = Structure<Transition>.Size;
85
86          public readonly ulong TransactionId;
87          public readonly UInt64Link Before;
88          public readonly UInt64Link After;
89          public readonly Timestamp Timestamp;
90
91          public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
            ↪ transactionId, UInt64Link before, UInt64Link after)
92          {
93              TransactionId = transactionId;
94              Before = before;
95              After = after;
96              Timestamp = uniqueTimestampFactory.Create();
97          }
98
99          public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
            ↪ transactionId, UInt64Link before)
100             : this(uniqueTimestampFactory, transactionId, before, default)
101         {
102         }
103
104         public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId)
105             : this(uniqueTimestampFactory, transactionId, default, default)
106         {
107         }
108
109         public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
            ↪ {After}";
110     }
111
112     /// <remarks>
113     /// Другие варианты реализации транзакций (атомарности):
114     ///     1. Разделение хранения значения связи ((Source Target) или (Source Linker
        ↪ Target)) и индексов.
115     ///     2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
        ↪ потребуется решить вопрос
116     ///        со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
        ↪ пересечениями идентификаторов.
117     ///
118     /// Где хранить промежуточный список транзакций?
119     ///
120     /// В оперативной памяти:
121     ///   Минусы:
122     ///     1. Может усложнить систему, если она будет функционировать самостоятельно,
123     ///     так как нужно отдельно выделять память под список трансформаций.
124     ///     2. Выделенной оперативной памяти может не хватить, в том случае,
125     ///     если транзакция использует слишком много трансформаций.
126     ///         -> Можно использовать жёсткий диск для слишком длинных транзакций.
127     ///         -> Максимальный размер списка трансформаций можно ограничить / задать
        ↪ константой.
128     ///     3. При подтверждении транзакции (Commit) все трансформации записываются разом
        ↪ создавая задержку.
129     ///
130     /// На жёстком диске:
131     ///   Минусы:
132     ///     1. Длительный отклик, на запись каждой трансформации.
133     ///     2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
134     ///         -> Это может решаться упаковкой/исключением дублирующих операций.
135     ///         -> Также это может решаться тем, что короткие транзакции вообще
136     ///            не будут записываться в случае отката.
137     ///     3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
        ↪ операции (трансформации)
138     ///        будут записаны в лог.
139     ///
140     /// </remarks>
141     public class Transaction : DisposableBase
142     {
143         private readonly Queue<Transition> _transitions;
144         private readonly UInt64LinksTransactionsLayer _layer;
145         public bool IsCommitted { get; private set; }
146         public bool IsReverted { get; private set; }
147
148         public Transaction(UInt64LinksTransactionsLayer layer)
149         {
150             _layer = layer;
151             if (_layer._currentTransactionId != 0)
```

```csharp
                    {
                        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)
                    {
                        if (!IsCommitted && !IsReverted)
                        {
                            Revert();
                        }
                        _layer.ResetCurrentTransation();
                    }
                }
            }

        public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);

        private readonly string _logAddress;
        private readonly FileStream _log;
        private readonly Queue<Transition> _transitions;
        private readonly UniqueTimestampFactory _uniqueTimestampFactory;
        private Task _transitionsPusher;
        private Transition _lastCommitedTransition;
        private ulong _currentTransactionId;
        private Queue<Transition> _currentTransactionTransitions;
        private Transaction _currentTransaction;
        private ulong _lastCommitedTransactionId;

        public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
```

```csharp
231             : base(links)
232         {
233             if (string.IsNullOrWhiteSpace(logAddress))
234             {
235                 throw new ArgumentNullException(nameof(logAddress));
236             }
237             // В первой строке файла хранится последняя закоммиченную транзакцию.
238             // При запуске это используется для проверки удачного закрытия файла лога.
239             // In the first line of the file the last committed transaction is stored.
240             // On startup, this is used to check that the log file is successfully closed.
241             var lastCommitedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
242             var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
243             if (!lastCommitedTransition.Equals(lastWrittenTransition))
244             {
245                 Dispose();
246                 throw new NotSupportedException("Database is damaged, autorecovery is not
    ↪    supported yet.");
247             }
248             if (lastCommitedTransition.Equals(default(Transition)))
249             {
250                 FileHelpers.WriteFirst(logAddress, lastCommitedTransition);
251             }
252             _lastCommitedTransition = lastCommitedTransition;
253             // TODO: Think about a better way to calculate or store this value
254             var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
255             _lastCommitedTransactionId = allTransitions.Max(x => x.TransactionId);
256             _uniqueTimestampFactory = new UniqueTimestampFactory();
257             _logAddress = logAddress;
258             _log = FileHelpers.Append(logAddress);
259             _transitions = new Queue<Transition>();
260             _transitionsPusher = new Task(TransitionsPusher);
261             _transitionsPusher.Start();
262         }
263
264         public IList<ulong> GetLinkValue(ulong link) => Links.GetLink(link);
265
266         public override ulong Create()
267         {
268             var createdLinkIndex = Links.Create();
269             var createdLink = new UInt64Link(Links.GetLink(createdLinkIndex));
270             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
    ↪    default, createdLink));
271             return createdLinkIndex;
272         }
273
274         public override ulong Update(IList<ulong> parts)
275         {
276             var linkIndex = parts[Constants.IndexPart];
277             var beforeLink = new UInt64Link(Links.GetLink(linkIndex));
278             linkIndex = Links.Update(parts);
279             var afterLink = new UInt64Link(Links.GetLink(linkIndex));
280             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
    ↪    beforeLink, afterLink));
281             return linkIndex;
282         }
283
284         public override void Delete(ulong link)
285         {
286             var deletedLink = new UInt64Link(Links.GetLink(link));
287             Links.Delete(link);
288             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
    ↪    deletedLink, default));
289         }
290
291         [MethodImpl(MethodImplOptions.AggressiveInlining)]
292         private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
    ↪    _transitions;
293
294         private void CommitTransition(Transition transition)
295         {
296             if (_currentTransaction != null)
297             {
298                 Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
299             }
300             var transitions = GetCurrentTransitions();
301             transitions.Enqueue(transition);
302         }
303
```

```csharp
304        private void RevertTransition(Transition transition)
305        {
306            if (transition.After.IsNull()) // Revert Deletion with Creation
307            {
308                Links.Create();
309            }
310            else if (transition.Before.IsNull()) // Revert Creation with Deletion
311            {
312                Links.Delete(transition.After.Index);
313            }
314            else // Revert Update
315            {
316                Links.Update(new[] { transition.After.Index, transition.Before.Source,
                    ↪  transition.Before.Target });
317            }
318        }
319
320        private void ResetCurrentTransation()
321        {
322            _currentTransactionId = 0;
323            _currentTransactionTransitions = null;
324            _currentTransaction = null;
325        }
326
327        private void PushTransitions()
328        {
329            if (_log == null || _transitions == null)
330            {
331                return;
332            }
333            for (var i = 0; i < _transitions.Count; i++)
334            {
335                var transition = _transitions.Dequeue();
336
337                _log.Write(transition);
338                _lastCommitedTransition = transition;
339            }
340        }
341
342        private void TransitionsPusher()
343        {
344            while (!IsDisposed && _transitionsPusher != null)
345            {
346                Thread.Sleep(DefaultPushDelay);
347                PushTransitions();
348            }
349        }
350
351        public Transaction BeginTransaction() => new Transaction(this);
352
353        private void DisposeTransitions()
354        {
355            try
356            {
357                var pusher = _transitionsPusher;
358                if (pusher != null)
359                {
360                    _transitionsPusher = null;
361                    pusher.Wait();
362                }
363                if (_transitions != null)
364                {
365                    PushTransitions();
366                }
367                _log.DisposeIfPossible();
368                FileHelpers.WriteFirst(_logAddress, _lastCommitedTransition);
369            }
370            catch
371            {
372            }
373        }
374
375        #region DisposalBase
376
377        protected override void Dispose(bool manual, bool wasDisposed)
378        {
379            if (!wasDisposed)
380            {
381                DisposeTransitions();
```

```
382            }
383            base.Dispose(manual, wasDisposed);
384        }
385
386        #endregion
387    }
388 }
```

./Platform.Data.Doublets/UnaryNumbers/AddressToUnaryNumberConverter.cs

```
1  using System.Collections.Generic;
2  using Platform.Interfaces;
3  using Platform.Reflection;
4  using Platform.Numbers;
5
6  namespace Platform.Data.Doublets.UnaryNumbers
7  {
8      public class AddressToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
        ↪  IConverter<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
11
12         private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;
13
14         public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
           ↪  powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
           ↪  powerOf2ToUnaryNumberConverter;
15
16         public TLink Convert(TLink sourceAddress)
17         {
18             var number = sourceAddress;
19             var nullConstant = Links.Constants.Null;
20             var one = Integer<TLink>.One;
21             var target = nullConstant;
22             for (int i = 0; !_equalityComparer.Equals(number, default) && i <
               ↪  Type<TLink>.BitsLength; i++)
23             {
24                 if (_equalityComparer.Equals(Arithmetic.And(number, one), one))
25                 {
26                     target = _equalityComparer.Equals(target, nullConstant)
27                         ? _powerOf2ToUnaryNumberConverter.Convert(i)
28                         : Links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
29                 }
30                 number = (Integer<TLink>)((ulong)(Integer<TLink>)number >> 1); // Should be
                   ↪  Bit.ShiftRight(number, 1)
31             }
32             return target;
33         }
34     }
35 }
```

./Platform.Data.Doublets/UnaryNumbers/LinkToItsFrequencyNumberConveter.cs

```
1  using System;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4
5  namespace Platform.Data.Doublets.UnaryNumbers
6  {
7      public class LinkToItsFrequencyNumberConveter<TLink> : LinksOperatorBase<TLink>,
        ↪  IConverter<Doublet<TLink>, TLink>
8      {
9          private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
10
11         private readonly IPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
12         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
13
14         public LinkToItsFrequencyNumberConveter(
15             ILinks<TLink> links,
16             IPropertyOperator<TLink, TLink> frequencyPropertyOperator,
17             IConverter<TLink> unaryNumberToAddressConverter)
18             : base(links)
19         {
20             _frequencyPropertyOperator = frequencyPropertyOperator;
21             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
22         }
23
24         public TLink Convert(Doublet<TLink> doublet)
25         {
26             var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
```

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

./Platform.Data.Doublets/UnaryNumbers/PowerOf2ToUnaryNumberConverter.cs

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

./Platform.Data.Doublets/UnaryNumbers/UnaryNumberToAddressAddOperationConverter.cs

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Interfaces;
4  using Platform.Numbers;
5
6  namespace Platform.Data.Doublets.UnaryNumbers
7  {
8      public class UnaryNumberToAddressAddOperationConverter<TLink> : LinksOperatorBase<TLink>,
         ↪  IConverter<TLink>
9      {
10          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
11
12          private Dictionary<TLink, TLink> _unaryToUInt64;
13          private readonly TLink _unaryOne;
14
15          public UnaryNumberToAddressAddOperationConverter(ILinks<TLink> links, TLink unaryOne)
16              : base(links)
17          {
18              _unaryOne = unaryOne;
19              InitUnaryToUInt64();
20          }
21
22          private void InitUnaryToUInt64()
23          {
```

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

./Platform.Data.Doublets/UnaryNumbers/UnaryNumberToAddressOrOperationConverter.cs

```
1  using System.Collections.Generic;
2  using Platform.Interfaces;
3  using Platform.Reflection;
4  using Platform.Numbers;
5  using System.Runtime.CompilerServices;
6
7  namespace Platform.Data.Doublets.UnaryNumbers
8  {
9      public class UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
   ↪  IConverter<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
   ↪  EqualityComparer<TLink>.Default;
12
13         private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;
14
15         public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
   ↪  TLink> powerOf2ToUnaryNumberConverter)
16             : base(links)
17         {
18             _unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
19             for (int i = 0; i < Type<TLink>.BitsLength; i++)
20             {
21                 _unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
22             }
23         }
```

```
24
25          public TLink Convert(TLink sourceNumber)
26          {
27              var nullConstant = Links.Constants.Null;
28              var source = sourceNumber;
29              var target = nullConstant;
30              if (!_equalityComparer.Equals(source, nullConstant))
31              {
32                  while (true)
33                  {
34                      if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
35                      {
36                          SetBit(ref target, powerOf2Index);
37                          break;
38                      }
39                      else
40                      {
41                          powerOf2Index = _unaryNumberPowerOf2Indicies[Links.GetSource(source)];
42                          SetBit(ref target, powerOf2Index);
43                          source = Links.GetTarget(source);
44                      }
45                  }
46              }
47              return target;
48          }
49
50          [MethodImpl(MethodImplOptions.AggressiveInlining)]
51          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))
52      }
53  }
```

./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  namespace Platform.Data.Doublets.Unicode
9  {
10     public class UnicodeMap
11     {
12         public static readonly ulong FirstCharLink = 1;
13         public static readonly ulong LastCharLink = FirstCharLink + char.MaxValue;
14         public static readonly ulong MapSize = 1 + char.MaxValue;
15
16         private readonly ILinks<ulong> _links;
17         private bool _initialized;
18
19         public UnicodeMap(ILinks<ulong> links) => _links = links;
20
21         public static UnicodeMap InitNew(ILinks<ulong> links)
22         {
23             var map = new UnicodeMap(links);
24             map.Init();
25             return map;
26         }
27
28         public void Init()
29         {
30             if (_initialized)
31             {
32                 return;
33             }
34             _initialized = true;
35             var firstLink = _links.CreatePoint();
36             if (firstLink != FirstCharLink)
37             {
38                 _links.Delete(firstLink);
39             }
40             else
41             {
42                 for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
43                 {
44                     // From NIL to It (NIL -> Character) transformation meaning, (or infinite
                     ↪ amount of NIL characters before actual Character)
45                     var createdLink = _links.CreatePoint();
```

```csharp
                    _links.Update(createdLink, firstLink, createdLink);
                    if (createdLink != i)
                    {
                        throw new InvalidOperationException("Unable to initialize UTF 16
                            table.");
                    }
                }
            }
        }

        // 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;
```

```csharp
                    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++;
                            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.Tests/ComparisonTests.cs
```csharp
using System;
using System.Collections.Generic;
using Xunit;
using Platform.Diagnostics;

namespace Platform.Data.Doublets.Tests
{
    public static class ComparisonTests
    {
```

```csharp
        protected class UInt64Comparer : IComparer<ulong>
        {
            public int Compare(ulong x, ulong y) => x.CompareTo(y);
        }

        private static int Compare(ulong x, ulong y) => x.CompareTo(y);

        [Fact]
        public static void GreaterOrEqualPerfomanceTest()
        {
            const int N = 1000000;

            ulong x = 10;
            ulong y = 500;

            bool result = false;

            var ts1 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = Compare(x, y) >= 0;
                }
            });

            var comparer1 = Comparer<ulong>.Default;

            var ts2 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = comparer1.Compare(x, y) >= 0;
                }
            });

            Func<ulong, ulong, int> compareReference = comparer1.Compare;

            var ts3 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = compareReference(x, y) >= 0;
                }
            });

            var comparer2 = new UInt64Comparer();

            var ts4 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = comparer2.Compare(x, y) >= 0;
                }
            });

            Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {result}");
        }
    }
}
```

./Platform.Data.Doublets.Tests/DoubletLinksTests.cs

```csharp
using System.Collections.Generic;
using Xunit;
using Platform.Reflection;
using Platform.Numbers;
using Platform.Memory;
using Platform.Scopes;
using Platform.Setters;
using Platform.Data.Doublets.ResizableDirectMemory;

namespace Platform.Data.Doublets.Tests
{
    public static class DoubletLinksTests
    {
        [Fact]
        public static void UInt64CRUDTest()
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪ ResizableDirectMemoryLinks<ulong>>>())
            {
```

```
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()
139     {
140         using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪ ResizableDirectMemoryLinks<byte>>>())
141         {
142             scope.Use<ILinks<byte>>().TestRawNumbersCRUDOperations();
143         }
144     }
145
146     private static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
147     {
148         // Constants
149         var constants = links.Constants;
150         var equalityComparer = EqualityComparer<T>.Default;
151
152         var h106E = new Hybrid<T>(106L, isExternal: true);
153         var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
154         var h108E = new Hybrid<T>(-108L);
155
156         Assert.Equal(106L, h106E.AbsoluteValue);
157         Assert.Equal(107L, h107E.AbsoluteValue);
158         Assert.Equal(108L, h108E.AbsoluteValue);
159
160         // Create Link (External -> External)
161         var linkAddress1 = links.Create();
162
163         links.Update(linkAddress1, h106E, h108E);
164
165         var link1 = new Link<T>(links.GetLink(linkAddress1));
166
167         Assert.True(equalityComparer.Equals(link1.Source, h106E));
168         Assert.True(equalityComparer.Equals(link1.Target, h108E));
169
```

```
170          // Create Link (Internal -> External)
171          var linkAddress2 = links.Create();
172
173          links.Update(linkAddress2, linkAddress1, h108E);
174
175          var link2 = new Link<T>(links.GetLink(linkAddress2));
176
177          Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
178          Assert.True(equalityComparer.Equals(link2.Target, h108E));
179
180          // Create Link (Internal -> Internal)
181          var linkAddress3 = links.Create();
182
183          links.Update(linkAddress3, linkAddress1, linkAddress2);
184
185          var link3 = new Link<T>(links.GetLink(linkAddress3));
186
187          Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
188          Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));
189
190          // Search for created link
191          var setter1 = new Setter<T>(constants.Null);
192          links.Each(h106E, h108E, setter1.SetAndReturnFalse);
193
194          Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));
195
196          // Search for nonexistent link
197          var setter2 = new Setter<T>(constants.Null);
198          links.Each(h106E, h107E, setter2.SetAndReturnFalse);
199
200          Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));
201
202          // Update link to reference null (prepare for delete)
203          var updated = links.Update(linkAddress3, constants.Null, constants.Null);
204
205          Assert.True(equalityComparer.Equals(updated, linkAddress3));
206
207          link3 = new Link<T>(links.GetLink(linkAddress3));
208
209          Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
210          Assert.True(equalityComparer.Equals(link3.Target, constants.Null));
211
212          // Delete link
213          links.Delete(linkAddress3);
214
215          Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Two));
216
217          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]
```

```csharp
public static void EqualsPerfomanceTest()
{
    const int N = 1000000;

    ulong x = 10;
    ulong y = 500;

    bool result = false;

    var ts1 = Performance.Measure(() =>
    {
        for (int i = 0; i < N; i++)
        {
            result = Equals1(x, y);
        }
    });

    var ts2 = Performance.Measure(() =>
    {
        for (int i = 0; i < N; i++)
        {
            result = Equals2(x, y);
        }
    });

    var ts3 = Performance.Measure(() =>
    {
        for (int i = 0; i < N; i++)
        {
            result = Equals3(x, y);
        }
    });

    var equalityComparer1 = EqualityComparer<ulong>.Default;

    var ts4 = Performance.Measure(() =>
    {
        for (int i = 0; i < N; i++)
        {
            result = equalityComparer1.Equals(x, y);
        }
    });

    var equalityComparer2 = new UInt64EqualityComparer();

    var ts5 = Performance.Measure(() =>
    {
        for (int i = 0; i < N; i++)
        {
            result = equalityComparer2.Equals(x, y);
        }
    });

    Func<ulong, ulong, bool> equalityComparer3 = equalityComparer2.Equals;

    var ts6 = Performance.Measure(() =>
    {
        for (int i = 0; i < N; i++)
        {
            result = equalityComparer3(x, y);
        }
    });

    var comparer = Comparer<ulong>.Default;

    var ts7 = Performance.Measure(() =>
    {
        for (int i = 0; i < N; i++)
        {
            result = comparer.Compare(x, y) == 0;
        }
    });

    Assert.True(ts2 < ts1);
    Assert.True(ts3 < ts2);
    Assert.True(ts5 < ts4);
    Assert.True(ts5 < ts6);

    Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {ts5} {ts6} {ts7} {result}");
```

```
103            }
104          }
105    }
```

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

```csharp
 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
44                      var created = 0;
45                      var deleted = 0;
46
47                      for (var i = 0; i < N; i++)
48                      {
49                          var linksCount = links.Count();
50
51                          var createPoint = random.NextBoolean();
52
53                          if (linksCount > 2 && createPoint)
54                          {
55                              var linksAddressRange = new Range<ulong>(1, linksCount);
56                              var source = random.NextUInt64(linksAddressRange);
57                              var target = random.NextUInt64(linksAddressRange); //-V3086
58
59                              var resultLink = links.CreateAndUpdate(source, target);
60                              if (resultLink > linksCount)
61                              {
62                                  created++;
63                              }
64                          }
65                          else
66                          {
67                              links.Create();
68                              created++;
69                          }
70                      }
71
72                      Assert.True(created == (int)links.Count());
73
74                      for (var i = 0; i < N; i++)
75                      {
```

```csharp
                        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))
            {
                var links = scope.Links;
                var l1 = links.Create();
                var l2 = links.Create();

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

                links.Delete(l1);

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

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

        [Fact]
        public static void TransactionAutoRevertedTest()
        {
            // Auto Reverted (Because no commit at transaction)
            using (var scope = new TempLinksTestScope(useLog: true))
            {
                var links = scope.Links;
                var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
                using (var transaction = transactionsLayer.BeginTransaction())
                {
                    var l1 = links.Create();
                    var l2 = links.Create();

                    links.Update(l2, l2, l1, l2);
                }
```

```
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();
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     {
```

```csharp
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)
341            {
342                Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
     ↪  yet.");
343            }
344
345            Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran↲
     ↪  sactionLogFilename);
346
347            File.Delete(tempDatabaseFilename);
348            File.Delete(tempTransactionLogFilename);
349        }
350
351        [Fact]
352        public static void Bug1Test()
353        {
354            var tempDatabaseFilename = Path.GetTempFileName();
355            var tempTransactionLogFilename = Path.GetTempFileName();
356
357            var itself = _constants.Itself;
358
359            // User Code Error (Autoreverted), some data saved
360            try
361            {
362                ulong l1;
363                ulong l2;
364
365                using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
     ↪  UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
     ↪  tempTransactionLogFilename))
366                using (var links = new UInt64Links(memoryAdapter))
367                {
368                    l1 = links.CreateAndUpdate(itself, itself);
```

```csharp
                        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()
            {
                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);
```

```
444                b = links.Update(b, a, c);
445                c = links.Update(c, a, b);
446
447                Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
448                Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
449                Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
450
451                Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
       ↪    "(5:(4:5 (6:5 4)) 6)");
452                Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
       ↪    "(6:(5:(4:5 6) 6) 4)");
453                Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
       ↪    "(4:(5:4 (6:5 4)) 6)");
454
455                // 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)"
456
457                Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
       ↪    "{{5}{5}{4}{6}}");
458                Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
       ↪    "{{5}{6}{6}{4}}");
459                Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
       ↪    "{{4}{5}{4}{6}}");
460            }
461        }
462
463        private static void DefaultFormatter(StringBuilder sb, ulong link)
464        {
465            sb.Append(link.ToString());
466        }
467
468        #endregion
469
470        #region Performance
471
472        /*
473        public static void RunAllPerformanceTests()
474        {
475            try
476            {
477                links.TestLinksInSteps();
478            }
479            catch (Exception ex)
480            {
481                ex.WriteToConsole();
482            }
483
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)
```

```
            {
                ex.WriteToConsole();
            }
        }*/

        /*
        public static void TestLinksInSteps()
        {
            const long gibibyte = 1024 * 1024 * 1024;
            const long mebibyte = 1024 * 1024;

            var totalLinksToCreate = gibibyte /
    ↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
            var linksStep = 102 * mebibyte /
    ↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;

            var creationMeasurements = new List<TimeSpan>();
            var searchMeasuremets = new List<TimeSpan>();
            var deletionMeasurements = new List<TimeSpan>();

            GetBaseRandomLoopOverhead(linksStep);
            GetBaseRandomLoopOverhead(linksStep);

            var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);

            ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);

            var loops = totalLinksToCreate / linksStep;

            for (int i = 0; i < loops; i++)
            {
                creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
                searchMeasuremets.Add(Measure(() => links.RunRandomSearches(linksStep)));

                Console.Write("\rC + S {0}/{1}", i + 1, loops);
            }

            ConsoleHelpers.Debug();

            for (int i = 0; i < loops; i++)
            {
                deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));

                Console.Write("\rD {0}/{1}", i + 1, loops);
            }

            ConsoleHelpers.Debug();

            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(() =>
            {
```

```csharp
            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);
        }
    }

    [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);
```

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

                    long counter = 0;

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

                    var sw = Stopwatch.StartNew();

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

                    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);
                }
            }

            // TODO: Заполнить базу данных перед тестом
            /*
            [Fact]
```

```csharp
        public void TestRandomSearchFixed()
        {
            var tempFilename = Path.GetTempFileName();

            using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    DefaultLinksSizeStep))
            {
                long iterations = 64 * 1024 * 1024 /
    Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;

                ulong counter = 0;
                var maxLink = links.Total;

                ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);

                var sw = Stopwatch.StartNew();

                for (var i = iterations; i > 0; i--)
                {
                    var source =
    RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
                    var target =
    RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);

                    counter += links.Search(source, target);
                }

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;

                ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
    Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
    counter);
            }

            File.Delete(tempFilename);
        }*/

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

                var maxLink = links.Count();

                var iterations = links.Count();

                ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
                    links.Count());

                var sw = Stopwatch.StartNew();

                for (var i = iterations; i > 0; i--)
                {
                    var linksAddressRange = new Range<ulong>(_constants.MinPossibleIndex,
                        maxLink);

                    var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
                    var target = RandomHelpers.Default.NextUInt64(linksAddressRange);

                    counter += links.SearchOrDefault(source, target);
                }

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;

                ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
                    Iterations per second), c: {3}",
                        iterations, elapsedTime, (long)iterationsPerSecond, counter);
            }
        }

        [Fact(Skip = "useless: O(0), was dependent on creation tests")]
        public static void TestEach()
```

```csharp
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;

                var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);

                ConsoleHelpers.Debug("Testing Each function.");

                var sw = Stopwatch.StartNew();

                links.Each(counter.IncrementAndReturnTrue);

                var elapsedTime = sw.Elapsed;

                var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;

                ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
                ↪   links per second)",
                    counter, elapsedTime, (long)linksPerSecond);
            }
        }

        /*
        [Fact]
        public static void TestForeach()
        {
            var tempFilename = Path.GetTempFileName();

            using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪   DefaultLinksSizeStep))
            {
                ulong counter = 0;

                ConsoleHelpers.Debug("Testing foreach through links.");

                var sw = Stopwatch.StartNew();

                //foreach (var link in links)
                //{
                //    counter++;
                //}

                var elapsedTime = sw.Elapsed;

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

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

            File.Delete(tempFilename);
        }
        */

        /*
        [Fact]
        public static void TestParallelForeach()
        {
            var tempFilename = Path.GetTempFileName();

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

                long counter = 0;

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

                var sw = Stopwatch.StartNew();

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

                var elapsedTime = sw.Elapsed;

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

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

                    File.Delete(tempFilename);
                }
                */

                [Fact(Skip = "performance test")]
                public static void Create64BillionLinks()
                {
                    using (var scope = new TempLinksTestScope())
                    {
                        var links = scope.Links;
                        var linksBeforeTest = links.Count();

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

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

                        var elapsedTime = Performance.Measure(() =>
                        {
                            for (long i = 0; i < linksToCreate; i++)
                            {
                                links.Create();
                            }
                        });

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

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

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

                [Fact(Skip = "performance test")]
                public static void Create64BillionLinksInParallel()
                {
                    using (var scope = new TempLinksTestScope())
                    {
                        var links = scope.Links;
                        var linksBeforeTest = links.Count();

                        var sw = Stopwatch.StartNew();

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

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

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

                        var elapsedTime = sw.Elapsed;

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

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

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

                        ConsoleHelpers.Debug("Deleting all links");

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

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

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

            #endregion
        }
    }
```

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

```csharp
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.UnaryNumbers;
15
16  namespace Platform.Data.Doublets.Tests
17  {
18      public static class OptimalVariantSequenceTests
19      {
20          private const string SequenceExample = "зеленела зелёная зелень";
21
22          [Fact]
23          public static void LinksBasedFrequencyStoredOptimalVariantSequenceTest()
24          {
25              using (var scope = new TempLinksTestScope(useSequences: false))
26              {
27                  var links = scope.Links;
28                  var constants = links.Constants;
29
30                  links.UseUnicode();
31
32                  var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);
33
34                  var meaningRoot = links.CreatePoint();
35                  var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
36                  var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
37                  var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
                        constants.Itself);
38
39                  var unaryNumberToAddressConveter = new
                        UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
40                  var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
41                  var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                        frequencyMarker, unaryOne, unaryNumberIncrementer);
42                  var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
                        frequencyPropertyMarker, frequencyMarker);
43                  var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
                        frequencyPropertyOperator, frequencyIncrementer);
44                  var linkToItsFrequencyNumberConverter = new
                        LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                        unaryNumberToAddressConveter);
45                  var sequenceToItsLocalElementLevelsConverter = new
                        SequenceToItsLocalElementLevelsConverter<ulong>(links,
                        linkToItsFrequencyNumberConverter);
46                  var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                        sequenceToItsLocalElementLevelsConverter);
47
48                  var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
                        Walker = new LeveledSequenceWalker<ulong>(links) });
49
50                  ExecuteTest(links, sequences, sequence,
                        sequenceToItsLocalElementLevelsConverter, index, optimalVariantConverter);
51              }
52          }
53
54          [Fact]
```

```csharp
            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(links, sequences, sequence,
                        sequenceToItsLocalElementLevelsConverter, index, optimalVariantConverter);
                }
            }

        private static void ExecuteTest(SynchronizedLinks<ulong> links, 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);

            Assert.True(sequence.SequenceEqual(readSequence1));
        }
    }
}
```

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

```csharp
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using Xunit;
using Platform.Data.Sequences;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Sequences;

namespace Platform.Data.Doublets.Tests
{
    public static class ReadSequenceTests
    {
        [Fact]
        public static void ReadSequenceTest()
        {
            const long sequenceLength = 2000;

            using (var scope = new TempLinksTestScope(useSequences: false))
            {
                var links = scope.Links;
                var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
                    Walker = new LeveledSequenceWalker<ulong>(links) });;;

                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

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

```csharp
36              {
37                  var link = memoryAdapter.Create();
38                  memoryAdapter.Delete(link);
39              }
40
41              [Fact]
42              public static void NonexistentReferencesHeapMemoryTest()
43              {
44                  using (var memory = new
                    ↪  HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
45                  using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
                    ↪  UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
46                  {
47                      memoryAdapter.TestNonexistentReferences();
48                  }
49              }
50
51              private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
52              {
53                  var link = memoryAdapter.Create();
54                  memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
55                  var resultLink = _constants.Null;
56                  memoryAdapter.Each(foundLink =>
57                  {
58                      resultLink = foundLink[_constants.IndexPart];
59                      return _constants.Break;
60                  }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
61                  Assert.True(resultLink == link);
62                  Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
63                  memoryAdapter.Delete(link);
64              }
65          }
66      }
```

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

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

```csharp
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();
46                  var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
47
48                  var sw2 = Stopwatch.StartNew();
49                  var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
50
51                  Assert.True(results1.Count > results2.Length);
52                  Assert.True(sw1.Elapsed > sw2.Elapsed);
53
54                  for (var i = 0; i < sequenceLength; i++)
55                  {
56                      links.Delete(sequence[i]);
57                  }
58
59                  Assert.True(links.Count() == 0);
60              }
61          }
62
63          //[Fact]
64          //public void CUDTest()
65          //{
66          //     var tempFilename = Path.GetTempFileName();
67
68          //     const long sequenceLength = 8;
69
70          //     const ulong itself = LinksConstants.Itself;
71
72          //     using (var memoryAdapter = new ResizableDirectMemoryLinks(tempFilename,
            ↪  DefaultLinksSizeStep))
73          //     using (var links = new Links(memoryAdapter))
74          //     {
75          //         var sequence = new ulong[sequenceLength];
76          //         for (var i = 0; i < sequenceLength; i++)
77          //             sequence[i] = links.Create(itself, itself);
78
```

```csharp
79
80      //          SequencesOptions o = new SequencesOptions();
81
82      // TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
83      //          o.
84
85
86      //          var sequences = new Sequences(links);
87
88      //          var sw1 = Stopwatch.StartNew();
89      //          var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
90
91      //          var sw2 = Stopwatch.StartNew();
92      //          var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
93
94      //          Assert.True(results1.Count > results2.Length);
95      //          Assert.True(sw1.Elapsed > sw2.Elapsed);
96
97      //          for (var i = 0; i < sequenceLength; i++)
98      //              links.Delete(sequence[i]);
99      //      }
100
101     //      File.Delete(tempFilename);
102     //}
103
104     [Fact]
105     public static void AllVariantsSearchTest()
106     {
107         const long sequenceLength = 8;
108
109         using (var scope = new TempLinksTestScope(useSequences: true))
110         {
111             var links = scope.Links;
112             var sequences = scope.Sequences;
113
114             var sequence = new ulong[sequenceLength];
115             for (var i = 0; i < sequenceLength; i++)
116             {
117                 sequence[i] = links.Create();
118             }
119
120             var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();
121
122             //for (int i = 0; i < createResults.Length; i++)
123             //    sequences.Create(createResults[i]);
124
125             var sw0 = Stopwatch.StartNew();
126             var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
127
128             var sw1 = Stopwatch.StartNew();
129             var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
130
131             var sw2 = Stopwatch.StartNew();
132             var searchResults2 = sequences.Each1(sequence); sw2.Stop();
133
134             var sw3 = Stopwatch.StartNew();
135             var searchResults3 = sequences.Each(sequence); sw3.Stop();
136
137             var intersection0 = createResults.Intersect(searchResults0).ToList();
138             Assert.True(intersection0.Count == searchResults0.Count);
139             Assert.True(intersection0.Count == createResults.Length);
140
141             var intersection1 = createResults.Intersect(searchResults1).ToList();
142             Assert.True(intersection1.Count == searchResults1.Count);
143             Assert.True(intersection1.Count == createResults.Length);
144
145             var intersection2 = createResults.Intersect(searchResults2).ToList();
146             Assert.True(intersection2.Count == searchResults2.Count);
147             Assert.True(intersection2.Count == createResults.Length);
148
149             var intersection3 = createResults.Intersect(searchResults3).ToList();
150             Assert.True(intersection3.Count == searchResults3.Count);
151             Assert.True(intersection3.Count == createResults.Length);
152
153             for (var i = 0; i < sequenceLength; i++)
154             {
155                 links.Delete(sequence[i]);
156             }
157         }
158     }
```

```csharp
        [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]);
                }
            }
        }

        [Fact]
        public static void AllPartialVariantsSearchTest()
        {
            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);

                //var createResultsStrings = createResults.Select(x => x + ": " +
                ↪   sequences.FormatSequence(x)).ToList();
                //Global.Trash = createResultsStrings;

                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();
```

```csharp
                        //var sw3 = Stopwatch.StartNew();
                        //var searchResults3 =
                        ↪    sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();

                        var sw4 = Stopwatch.StartNew();
                        var searchResults4 =
                        ↪    sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();

                        //Global.Trash = searchResults3;

                        //var searchResults1Strings = searchResults1.Select(x => x + ": " +
                        ↪    sequences.FormatSequence(x)).ToList();
                        //Global.Trash = searchResults1Strings;

                        var intersection1 = createResults.Intersect(searchResults1).ToList();
                        Assert.True(intersection1.Count == createResults.Length);

                        var intersection2 = createResults.Intersect(searchResults2).ToList();
                        Assert.True(intersection2.Count == createResults.Length);

                        var intersection4 = createResults.Intersect(searchResults4).ToList();
                        Assert.True(intersection4.Count == createResults.Length);

                        for (var i = 0; i < sequenceLength; i++)
                        {
                            links.Delete(sequence[i]);
                        }
                    }
                }

        [Fact]
        public static void BalancedPartialVariantsSearchTest()
        {
            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 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;
```

```csharp
308            using (var scope = new TempLinksTestScope(useSequences: true))
309            {
310                var links = scope.Links;
311                var sequences = scope.Sequences;
312
313                var e1 = links.Create();
314                var e2 = links.Create();
315
316                var sequence = new[]
317                {
318                    e1, e2, e1, e2 // mama / papa
319                };
320
321                var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
322
323                var balancedVariant = balancedVariantConverter.Convert(sequence);
324
325                // 1: [1]
326                // 2: [2]
327                // 3: [1,2]
328                // 4: [1,2,1,2]
329
330                var doublet = links.GetSource(balancedVariant);
331
332                var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);
333
334                Assert.True(matchedSequences1.Count == 0);
335
336                var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);
337
338                Assert.True(matchedSequences2.Count == 0);
339
340                var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);
341
342                Assert.True(matchedSequences3.Count == 0);
343
344                var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
345
346                Assert.Contains(doublet, matchedSequences4);
347                Assert.Contains(balancedVariant, matchedSequences4);
348
349                for (var i = 0; i < sequence.Length; i++)
350                {
351                    links.Delete(sequence[i]);
352                }
353            }
354        }
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)
404
405 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?
406
407 [![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)
408
409 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?
410
411 [![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)
412
413 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?
414
415 [![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)
416

417 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?
418
419 [![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)
420
421 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?
422
423 [![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.c
     → om/Konard/LinksPlatform/master/doc/Intro/10.png)
424
425 На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом
     → рекурсии или фрактала?
426
427 [![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.githubuserc
     → ontent.com/Konard/LinksPlatform/master/doc/Intro/11.png)
428
429 Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы? Буквы?
     → Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?
430
431 [![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)
432
433 ...
434
435 [![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-anima
     → tion-500.gif
     → ""анимация"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro
     → -animation-500.gif)";
436
437
438   private static readonly string _exampleLoremIpsumText =
439   @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
                 → incididunt ut labore et dolore magna aliqua.
440 Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
     → consequat.";
441
442   [Fact]
443   public static void CompressionTest()
444   {
445   using (var scope = new TempLinksTestScope(useSequences: true))
446   {
447   var links = scope.Links;
448   var sequences = scope.Sequences;
449
450   var e1 = links.Create();
451   var e2 = links.Create();
452
453   var sequence = new[]
454   {
455   e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
456   };
457
458   var balancedVariantConverter = new BalancedVariantConverter<ulong>(links.Unsync);
459   var totalSequenceSymbolFrequencyCounter = new
                     → TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
460   var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync,
                     → totalSequenceSymbolFrequencyCounter);
461   var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
                     → balancedVariantConverter, doubletFrequenciesCache);

```
462
463                var compressedVariant = compressingConverter.Convert(sequence);
464
465                // 1: [1]         (1->1) point
466                // 2: [2]         (2->2) point
467                // 3: [1,2]       (1->2) doublet
468                // 4: [1,2,1,2] (3->3) doublet
469
470                Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
471                Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
472                Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
473                Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);
474
475                var source = _constants.SourcePart;
476                var target = _constants.TargetPart;
477
478                Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
479                Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
480                Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
481                Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);
482
483                // 4 - length of sequence
484                Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0)
                   ↪  == sequence[0]);
485                Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
                   ↪  == sequence[1]);
486                Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
                   ↪  == sequence[2]);
487                Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
                   ↪  == sequence[3]);
488            }
489        }
490
491        [Fact]
492        public static void CompressionEfficiencyTest()
493        {
494            var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
                   ↪  StringSplitOptions.RemoveEmptyEntries);
495            var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
496            var totalCharacters = arrays.Select(x => x.Length).Sum();
497
498            using (var scope1 = new TempLinksTestScope(useSequences: true))
499            using (var scope2 = new TempLinksTestScope(useSequences: true))
500            using (var scope3 = new TempLinksTestScope(useSequences: true))
501            {
502                scope1.Links.Unsync.UseUnicode();
503                scope2.Links.Unsync.UseUnicode();
504                scope3.Links.Unsync.UseUnicode();
505
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 unaryNumberToAddressConveter = new
                   ↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
523                //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
                   ↪  unaryOne);
524                //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                   ↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
```

```csharp
            //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
            ↪    frequencyPropertyMarker, frequencyMarker);
            //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
            ↪    frequencyPropertyOperator, frequencyIncrementer);
            //var linkToItsFrequencyNumberConverter = new
            ↪    LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
            ↪    unaryNumberToAddressConveter);

            var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
            ↪    totalSequenceSymbolFrequencyCounter);

            var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque⌋
            ↪    ncyNumberConverter<ulong>(linkFrequenciesCache3);

            var sequenceToItsLocalElementLevelsConverter = new
            ↪    SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
            ↪    linkToItsFrequencyNumberConverter);
            var optimalVariantConverter = new
            ↪    OptimalVariantConverter<ulong>(scope3.Links.Unsync,
            ↪    sequenceToItsLocalElementLevelsConverter);

            var compressed1 = new ulong[arrays.Length];
            var compressed2 = new ulong[arrays.Length];
            var compressed3 = new ulong[arrays.Length];

            var START = 0;
            var END = arrays.Length;

            //for (int i = START; i < END; i++)
            //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);

            var initialCount1 = scope2.Links.Unsync.Count();

            var sw1 = Stopwatch.StartNew();

            for (int i = START; i < END; i++)
            {
                linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
                compressed1[i] = compressor1.Convert(arrays[i]);
            }

            var elapsed1 = sw1.Elapsed;

            var balancedVariantConverter2 = new
            ↪    BalancedVariantConverter<ulong>(scope2.Links.Unsync);

            var initialCount2 = scope2.Links.Unsync.Count();

            var sw2 = Stopwatch.StartNew();

            for (int i = START; i < END; i++)
            {
                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);
```

```csharp
            // 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);

            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();
        }
    }
```

```csharp
        [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
                //    {
                //        // TODO: Find a solution for this case
                //    }
                //}

                for (int i = START; i < END; i++)
                {
                    var first = compressor1.Create(arrays[i]);
                    var second = compressor1.Create(arrays[i]);

                    if (first == second)
                    {
                        compressed1[i] = first;
                    }
                    else
                    {
                        // TODO: Find a solution for this case
                    }
                }

                var elapsed1 = sw1.Elapsed;

                var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);

                var sw2 = Stopwatch.StartNew();

                for (int i = START; i < END; i++)
                {
                    var first = balancedVariantConverter.Convert(arrays[i]);
```

```csharp
                    var second = balancedVariantConverter.Convert(arrays[i]);

                    if (first == second)
                    {
                        compressed2[i] = first;
                    }
                }

                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)
                    {
                        var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                        ↪  scope1.Links);

                        var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                        ↪  scope2.Links);

                        //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
                        ↪  link.IsPartialPoint());
                        //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
                        ↪  link.IsPartialPoint());

                        //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
                        ↪  arrays[i].Length > 3)
                        //    Assert.False(structure1 == structure2);

                        Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
                    }
                }

                Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
                Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);

                Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
                ↪  totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
                ↪  totalCharacters}");

                Assert.True(scope1.Links.Count() <= scope2.Links.Count());

                //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();
```

```csharp
                    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)
                        {
                            var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                            →   scope1.Links);

                            var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                            →   scope2.Links);

                            Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
                        }
                    }

                    Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
                    Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);

                    Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
                    →   totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
                    →   totalCharacters}");

                    // Can be worse than balanced variant
                    //Assert.True(scope1.Links.Count() <= scope2.Links.Count());

                    //compressor1.ValidateFrequencies();
                }
            }

        [Fact]
        public static void AllTreeBreakDownAtSequencesCreationBugTest()
        {
```

```csharp
            // Made out of AllPossibleConnectionsTest test.

            //const long sequenceLength = 5; //100% bug
            const long sequenceLength = 4; //100% bug
            //const long sequenceLength = 3; //100% _no_bug (ok)

            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);

                Global.Trash = createResults;

                for (var i = 0; i < sequenceLength; i++)
                {
                    links.Delete(sequence[i]);
                }
            }
        }

        [Fact]
        public static void AllPossibleConnectionsTest()
        {
            const long sequenceLength = 5;

            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);
                var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());

                for (var i = 0; i < 1; i++)
                {
                    var sw1 = Stopwatch.StartNew();
                    var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();

                    var sw2 = Stopwatch.StartNew();
                    var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();

                    var sw3 = Stopwatch.StartNew();
                    var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();

                    var sw4 = Stopwatch.StartNew();
                    var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();

                    Global.Trash = searchResults3;
                    Global.Trash = searchResults4; //-V3008

                    var intersection1 = createResults.Intersect(searchResults1).ToList();
                    Assert.True(intersection1.Count == createResults.Length);

                    var intersection2 = reverseResults.Intersect(searchResults1).ToList();
                    Assert.True(intersection2.Count == reverseResults.Length);

                    var intersection0 = searchResults1.Intersect(searchResults2).ToList();
                    Assert.True(intersection0.Count == searchResults2.Count);

                    var intersection3 = searchResults2.Intersect(searchResults3).ToList();
                    Assert.True(intersection3.Count == searchResults3.Count);

                    var intersection4 = searchResults3.Intersect(searchResults4).ToList();
                    Assert.True(intersection4.Count == searchResults4.Count);
                }
```

```
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 =
     ↪  sequences.CreateAllVariants2(sequence.Reverse().ToArray());
977
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    }
```

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

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

```
27              TempTransactionLogFilename = Path.GetTempFileName();
28
29              var coreMemoryAdapter = new UInt64ResizableDirectMemoryLinks(TempFilename);
30
31              MemoryAdapter = useLog ? (ILinks<ulong>)new
     ↪         UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
     ↪         coreMemoryAdapter;
32
33              Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
34              if (useSequences)
35              {
36                  Sequences = new Sequences.Sequences(Links, sequencesOptions);
37              }
38          }
39
40          protected override void Dispose(bool manual, bool wasDisposed)
41          {
42              if (!wasDisposed)
43              {
44                  Links.Unsync.DisposeIfPossible();
45                  if (_deleteFiles)
46                  {
47                      DeleteFiles();
48                  }
49              }
50          }
51
52          public void DeleteFiles()
53          {
54              File.Delete(TempFilename);
55              File.Delete(TempTransactionLogFilename);
56          }
57      }
58  }
```

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

```
1  using Xunit;
2  using Platform.Random;
3  using Platform.Data.Doublets.UnaryNumbers;
4
5  namespace Platform.Data.Doublets.Tests
6  {
7      public static class UnaryNumberConvertersTests
8      {
9          [Fact]
10          public static void ConvertersTest()
11          {
12              using (var scope = new TempLinksTestScope())
13              {
14                  const int N = 10;
15                  var links = scope.Links;
16                  var meaningRoot = links.CreatePoint();
17                  var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
18                  var powerOf2ToUnaryNumberConverter = new
     ↪             PowerOf2ToUnaryNumberConverter<ulong>(links, one);
19                  var toUnaryNumberConverter = new AddressToUnaryNumberConverter<ulong>(links,
     ↪             powerOf2ToUnaryNumberConverter);
20                  var random = new System.Random(0);
21                  ulong[] numbers = new ulong[N];
22                  ulong[] unaryNumbers = new ulong[N];
23                  for (int i = 0; i < N; i++)
24                  {
25                      numbers[i] = random.NextUInt64();
26                      unaryNumbers[i] = toUnaryNumberConverter.Convert(numbers[i]);
27                  }
28                  var fromUnaryNumberConverterUsingOrOperation = new
     ↪             UnaryNumberToAddressOrOperationConverter<ulong>(links,
     ↪             powerOf2ToUnaryNumberConverter);
29                  var fromUnaryNumberConverterUsingAddOperation = new
     ↪             UnaryNumberToAddressAddOperationConverter<ulong>(links, one);
30                  for (int i = 0; i < N; i++)
31                  {
32                      Assert.Equal(numbers[i],
     ↪                 fromUnaryNumberConverterUsingOrOperation.Convert(unaryNumbers[i]));
33                      Assert.Equal(numbers[i],
     ↪                 fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));
34                  }
35              }
```

```
36            }
37        }
38    }
```

Index