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

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

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)
        {
            // Use Facade (the last decorator) to ensure recursion working correctly
            Facade.MergeUsages(oldLinkAddress, newLinkAddress);
            return base.ResolveAddressChangeConflict(oldLinkAddress, newLinkAddress);
        }
    }
}
```

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

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

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)
        {
            // Use Facade (the last decorator) to ensure recursion working correctly
            Facade.DeleteAllUsages(linkIndex);
            Links.Delete(linkIndex);
        }
    }
}
```

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

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

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

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

        public ILinks<TLink> _facade;

        public ILinks<TLink> Facade
        {
            get => _facade;
            private set
            {
                _facade = value;
                if (Links is LinksDecoratorBase<TLink> decorator)
                {
                    decorator.Facade = value;
                }
            }
        }

        protected LinksDecoratorBase(ILinks<TLink> links) : base(links)
        {
            Constants = links.Constants;
            Facade = this;
        }

        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

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

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

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);

        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);

        protected override bool AllowMultipleDisposeCalls => true;

        protected override void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                Links.DisposeIfPossible();
            }
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Decorators
{
    // TODO: Make LinksExternalReferenceValidator. A layer that checks each link to exist or to
    ↪  be external (hybrid link's raw number).
    public class LinksInnerReferenceExistenceValidator<TLink> : LinksDecoratorBase<TLink>
    {
        public LinksInnerReferenceExistenceValidator(ILinks<TLink> links) : base(links) { }

        public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
        {
            Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
            return Links.Each(handler, restrictions);
        }

        public override TLink Update(IList<TLink> restrictions)
        {
            // TODO: Possible values: null, ExistentLink or NonExistentHybrid(ExternalReference)
            Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
            return Links.Update(restrictions);
        }
```

```

        public override void Delete(TLink link)
        {
            Links.EnsureLinkExists(link, nameof(link));
            Links.Delete(link);
        }
    }
}
```

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

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

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

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

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

        public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
        {
            var constants = Constants;
            var itselfConstant = constants.Itself;
            var indexPartConstant = constants.IndexPart;
            var sourcePartConstant = constants.SourcePart;
            var targetPartConstant = constants.TargetPart;
            var restrictionsCount = restrictions.Count;
            if (!_equalityComparer.Equals(constants.Any, itselfConstant)
             && (((restrictionsCount > indexPartConstant) &&
              ↪  _equalityComparer.Equals(restrictions[indexPartConstant], itselfConstant))
             || ((restrictionsCount > sourcePartConstant) &&
              ↪  _equalityComparer.Equals(restrictions[sourcePartConstant], itselfConstant))
             || ((restrictionsCount > targetPartConstant) &&
              ↪  _equalityComparer.Equals(restrictions[targetPartConstant], itselfConstant))))
            {
                // Itself constant is not supported for Each method right now, skipping execution
                return constants.Continue;
            }
            return Links.Each(handler, restrictions);
        }

        public override TLink Update(IList<TLink> restrictions) =>
        ↪  Links.Update(Links.ResolveConstantAsSelfReference(Constants.Itself, restrictions));
    }
}
```

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

```
using System.Collections.Generic;

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

namespace Platform.Data.Doublets.Decorators
{
    /// <remarks>
    /// Not practical if newSource and newTarget are too big.
    /// To be able to use practical version we should allow to create link at any specific
    ↪  location inside ResizableDirectMemoryLinks.
    /// This in turn will require to implement not a list of empty links, but a list of ranges
    ↪  to store it more efficiently.
    /// </remarks>
    public class LinksNonExistentDependenciesCreator<TLink> : LinksDecoratorBase<TLink>
    {
        public LinksNonExistentDependenciesCreator(ILinks<TLink> links) : base(links) { }

        public override TLink Update(IList<TLink> restrictions)
        {
            var constants = Constants;
            Links.EnsureCreated(restrictions[constants.SourcePart],
            ↪  restrictions[constants.TargetPart]);
            return Links.Update(restrictions);
        }
    }
}
```

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

```
using System.Collections.Generic;

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

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

        public override TLink Create()
        {
            var link = Links.Create();
            return Links.Update(link, link, link);
        }

        public override TLink Update(IList<TLink> restrictions) =>
        ↪ Links.Update(Links.ResolveConstantAsSelfReference(Constants.Null, restrictions));
    }
}
```

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

```
using System.Collections.Generic;

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

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

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

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

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

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

```
using System.Collections.Generic;

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

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

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

```
    }
}
```

./Platform.Data.Doublets/Decorators/LinksUsagesValidator.cs
```
using System.Collections.Generic;

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

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

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

        public override void Delete(TLink link)
        {
            Links.EnsureNoUsages(link);
            Links.Delete(link);
        }
    }
}
```

./Platform.Data.Doublets/Decorators/NonNullContentsLinkDeletionResolver.cs
```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

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

./Platform.Data.Doublets/Decorators/UInt64Links.cs
```
using System;
using System.Collections.Generic;
using Platform.Collections;

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

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>
    {
```

```
29          public UInt64Links(ILinks<ulong> links) : base(links) { }
30
31          public override ulong Each(Func<IList<ulong>, ulong> handler, IList<ulong> restrictions)
32          {
33              this.EnsureLinkIsAnyOrExists(restrictions);
34              return Links.Each(handler, restrictions);
35          }
36
37          public override ulong Create() => Links.CreatePoint();
38
39          public override ulong Update(IList<ulong> restrictions)
40          {
41              var constants = Constants;
42              var nullConstant = constants.Null;
43              if (restrictions.IsNullOrEmpty())
44              {
45                  return nullConstant;
46              }
47              // TODO: Looks like this is a common type of exceptions linked with restrictions
                ↪  support
48              if (restrictions.Count != 3)
49              {
50                  throw new NotSupportedException();
51              }
52              var indexPartConstant = constants.IndexPart;
53              var updatedLink = restrictions[indexPartConstant];
54              this.EnsureLinkExists(updatedLink,
                ↪  $"{nameof(restrictions)}[{nameof(indexPartConstant)}]");
55              var sourcePartConstant = constants.SourcePart;
56              var newSource = restrictions[sourcePartConstant];
57              this.EnsureLinkIsItselfOrExists(newSource,
                ↪  $"{nameof(restrictions)}[{nameof(sourcePartConstant)}]");
58              var targetPartConstant = constants.TargetPart;
59              var newTarget = restrictions[targetPartConstant];
60              this.EnsureLinkIsItselfOrExists(newTarget,
                ↪  $"{nameof(restrictions)}[{nameof(targetPartConstant)}]");
61              var existedLink = nullConstant;
62              var itselfConstant = constants.Itself;
63              if (newSource != itselfConstant && newTarget != itselfConstant)
64              {
65                  existedLink = this.SearchOrDefault(newSource, newTarget);
66              }
67              if (existedLink == nullConstant)
68              {
69                  var before = Links.GetLink(updatedLink);
70                  if (before[sourcePartConstant] != newSource || before[targetPartConstant] !=
                    ↪  newTarget)
71                  {
72                      Links.Update(updatedLink, newSource == itselfConstant ? updatedLink :
                        ↪  newSource,
73                                                newTarget == itselfConstant ? updatedLink :
                                                  ↪  newTarget);
74                  }
75                  return updatedLink;
76              }
77              else
78              {
79                  return this.MergeAndDelete(updatedLink, existedLink);
80              }
81          }
82
83          public override void Delete(ulong linkIndex)
84          {
85              Links.EnsureLinkExists(linkIndex);
86              Links.EnforceResetValues(linkIndex);
87              this.DeleteAllUsages(linkIndex);
88              Links.Delete(linkIndex);
89          }
90      }
91  }
```

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

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

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

```
////                    newValue[Constants.SourcePart] =
↪  Equals(substitution[Constants.SourcePart], Constants.Itself) ?
↪  matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
////                    newValue[Constants.TargetPart] =
↪  Equals(substitution[Constants.TargetPart], Constants.Itself) ?
↪  matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
////                    var matchDecision = matchedHandler(matchedLink, newValue);
////                    if (Equals(matchDecision, Constants.Break))
////                        return false;
////                    if (!Equals(matchDecision, Constants.Skip))
////                        transitions.Add(new Transition(matchedLink, newValue));
////                    return true;
////                };
////                if (!Memory.Each(handler, restriction))
////                    return Constants.Break;
////            }
////        }
////        else
////        {
////            Func<T, bool> handler = link =>
////            {
////                var matchedLink = Memory.GetLinkValue(link);
////                var matchDecision = matchedHandler(matchedLink, matchedLink);
////                return !Equals(matchDecision, Constants.Break);
////            };
////            if (!Memory.Each(handler, restriction))
////                return Constants.Break;
////        }
////    }
////    else
////    {
////        if (substitution != null)
////        {
////            transitions = new List<IList<T>>();
////            Func<T, bool> handler = link =>
////            {
////                var matchedLink = Memory.GetLinkValue(link);
////                transitions.Add(matchedLink);
////                return true;
////            };
////            if (!Memory.Each(handler, restriction))
////                return Constants.Break;
////        }
////        else
////        {
////            return Constants.Continue;
////        }
////    }
////}
////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))
```

```
144             //        return Constants.Break;
145             //}
146             //else // Update
147             //{
148             //    //List<IList<T>> matchedLinks = null;
149             //    if (matchedHandler != null)
150             //    {
151             //        matchedLinks = new List<IList<T>>();
152             //        Func<T, bool> handler = link =>
153             //        {
154             //            var matchedLink = Memory.GetLinkValue(link);
155             //            var matchDecision = matchedHandler(matchedLink);
156             //            if (Equals(matchDecision, Constants.Break))
157             //                return false;
158             //            if (!Equals(matchDecision, Constants.Skip))
159             //                matchedLinks.Add(matchedLink);
160             //            return true;
161             //        };
162             //        if (!Memory.Each(handler, restriction))
163             //            return Constants.Break;
164             //    }
165             //    if (!matchedLinks.IsNullOrEmpty())
166             //    {
167             //        var totalMatchedLinks = matchedLinks.Count;
168             //        for (var i = 0; i < totalMatchedLinks; i++)
169             //        {
170             //            var matchedLink = matchedLinks[i];
171             //            if (substitutedHandler != null)
172             //            {
173             //                var newValue = new List<T>(); // TODO: Prepare value to update here
174             //                // TODO: Decide is it actually needed to use Before and After
                ↪  substitution handling.
175             //                var substitutedDecision = substitutedHandler(matchedLink,
                ↪  newValue);
176             //                if (Equals(substitutedDecision, Constants.Break))
177             //                    return Constants.Break;
178             //                if (Equals(substitutedDecision, Constants.Continue))
179             //                {
180             //                    // Actual update here
181             //                    Memory.SetLinkValue(newValue);
182             //                }
183             //                if (Equals(substitutedDecision, Constants.Skip))
184             //                {
185             //                    // Cancel the update. TODO: decide use separate Cancel
                ↪  constant or Skip is enough?
186             //                }
187             //            }
188             //        }
189             //    }
190             //}
191             return Constants.Continue;
192         }
193 
194         public TLink Trigger(IList<TLink> patternOrCondition, Func<IList<TLink>, TLink>
            ↪  matchHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
            ↪  substitutionHandler)
195         {
196             if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
197             {
198                 return Constants.Continue;
199             }
200             else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO:
                ↪  Check if it is a correct condition
201             {
202                 // Or it only applies to trigger without matchHandler.
203                 throw new NotImplementedException();
204             }
205             else if (!substitution.IsNullOrEmpty()) // Creation
206             {
207                 var before = ArrayPool<TLink>.Empty;
208                 // Что должно означать False здесь? Остановиться (перестать идти) или пропустить
                    ↪  (пройти мимо) или пустить (взять)?
209                 if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                    ↪  Constants.Break))
210                 {
211                     return Constants.Break;
212                 }
213                 var after = (IList<TLink>)substitution.ToArray();
```

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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

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

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

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

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

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

./Platform.Data.Doublets/Hybrid.cs

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

```
            var unsignedMethod = typeof(To).GetTypeInfo().GetMethod("Unsigned", new[] { signedVersion });
            emiter.Call(unsignedMethod);
            emiter.Return();
        });
    }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

./Platform.Data.Doublets/ILinks.cs

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

./Platform.Data.Doublets/ILinksExtensions.cs

```
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System.Linq;
5  using System.Runtime.CompilerServices;
6  using Platform.Ranges;
7  using Platform.Collections.Arrays;
8  using Platform.Numbers;
9  using Platform.Random;
10 using Platform.Setters;
11 using Platform.Data.Exceptions;
12 using Platform.Data.Doublets.Decorators;
13
14 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16 namespace Platform.Data.Doublets
17 {
18     public static class ILinksExtensions
19     {
20         public static void RunRandomCreations<TLink>(this ILinks<TLink> links, long
            ↪ amountOfCreations)
21         {
22             for (long i = 0; i < amountOfCreations; i++)
23             {
24                 var linksAddressRange = new Range<ulong>(0, (Integer<TLink>)links.Count());
25                 Integer<TLink> source = RandomHelpers.Default.NextUInt64(linksAddressRange);
26                 Integer<TLink> target = RandomHelpers.Default.NextUInt64(linksAddressRange);
27                 links.CreateAndUpdate(source, target);
28             }
29         }
30
31         public static void RunRandomSearches<TLink>(this ILinks<TLink> links, long
            ↪ amountOfSearches)
32         {
33             for (long i = 0; i < amountOfSearches; i++)
34             {
35                 var linkAddressRange = new Range<ulong>(1, (Integer<TLink>)links.Count());
36                 Integer<TLink> source = RandomHelpers.Default.NextUInt64(linkAddressRange);
37                 Integer<TLink> target = RandomHelpers.Default.NextUInt64(linkAddressRange);
38                 links.SearchOrDefault(source, target);
39             }
40         }
41
42         public static void RunRandomDeletions<TLink>(this ILinks<TLink> links, long
            ↪ amountOfDeletions)
43         {
44             var min = (ulong)amountOfDeletions > (Integer<TLink>)links.Count() ? 1 :
                ↪ (Integer<TLink>)links.Count() - (ulong)amountOfDeletions;
45             for (long i = 0; i < amountOfDeletions; i++)
46             {
47                 var linksAddressRange = new Range<ulong>(min, (Integer<TLink>)links.Count());
48                 Integer<TLink> link = RandomHelpers.Default.NextUInt64(linksAddressRange);
49                 links.Delete(link);
```

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

```
124             {
125                 var next = path[i];
126                 var values = links.GetLink(current);
127                 var source = values[constants.SourcePart];
128                 var target = values[constants.TargetPart];
129                 if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
                    ↪ next))
130                 {
131                     //throw new InvalidOperationException(string.Format("Невозможно выбрать
                        ↪ путь, так как и Source и Target совпадают с элементом пути {0}.", next));
132                     return false;
133                 }
134                 if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
                    ↪ target))
135                 {
136                     //throw new InvalidOperationException(string.Format("Невозможно продолжить
                        ↪ путь через элемент пути {0}", next));
137                     return false;
138                 }
139                 current = next;
140             }
141             return true;
142         }
143 
144         /// <remarks>
145         /// Может потребовать дополнительного стека для PathElement's при использовании
            ↪ SequenceWalker.
146         /// </remarks>
147         public static TLink GetByKeys<TLink>(this ILinks<TLink> links, TLink root, params int[]
            ↪ path)
148         {
149             links.EnsureLinkExists(root, "root");
150             var currentLink = root;
151             for (var i = 0; i < path.Length; i++)
152             {
153                 currentLink = links.GetLink(currentLink)[path[i]];
154             }
155             return currentLink;
156         }
157 
158         public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this ILinks<TLink>
            ↪ links, TLink root, ulong size, ulong index)
159         {
160             var constants = links.Constants;
161             var source = constants.SourcePart;
162             var target = constants.TargetPart;
163             if (!Platform.Numbers.Math.IsPowerOfTwo(size))
164             {
165                 throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes other
                    ↪ than powers of two are not supported.");
166             }
167             var path = new BitArray(BitConverter.GetBytes(index));
168             var length = Bit.GetLowestPosition(size);
169             links.EnsureLinkExists(root, "root");
170             var currentLink = root;
171             for (var i = length - 1; i >= 0; i--)
172             {
173                 currentLink = links.GetLink(currentLink)[path[i] ? target : source];
174             }
175             return currentLink;
176         }
177 
178         #endregion
179 
180         /// <summary>
181         /// Возвращает индекс указанной связи.
182         /// </summary>
183         /// <param name="links">Хранилище связей.</param>
184         /// <param name="link">Связь представленная списком, состоящим из её адреса и
            ↪ содержимого.</param>
185         /// <returns>Индекс начальной связи для указанной связи.</returns>
186         [MethodImpl(MethodImplOptions.AggressiveInlining)]
187         public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
            ↪ link[links.Constants.IndexPart];
188 
189         /// <summary>
190         /// Возвращает индекс начальной (Source) связи для указанной связи.
191         /// </summary>
```

```
192         /// <param name="links">Хранилище связей.</param>
193         /// <param name="link">Индекс связи.</param>
194         /// <returns>Индекс начальной связи для указанной связи.</returns>
195         [MethodImpl(MethodImplOptions.AggressiveInlining)]
196         public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink link) =>
            ↪ links.GetLink(link)[links.Constants.SourcePart];
197
198         /// <summary>
199         /// Возвращает индекс начальной (Source) связи для указанной связи.
200         /// </summary>
201         /// <param name="links">Хранилище связей.</param>
202         /// <param name="link">Связь представленная списком, состоящим из её адреса и
            ↪ содержимого.</param>
203         /// <returns>Индекс начальной связи для указанной связи.</returns>
204         [MethodImpl(MethodImplOptions.AggressiveInlining)]
205         public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
            ↪ link[links.Constants.SourcePart];
206
207         /// <summary>
208         /// Возвращает индекс конечной (Target) связи для указанной связи.
209         /// </summary>
210         /// <param name="links">Хранилище связей.</param>
211         /// <param name="link">Индекс связи.</param>
212         /// <returns>Индекс конечной связи для указанной связи.</returns>
213         [MethodImpl(MethodImplOptions.AggressiveInlining)]
214         public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
            ↪ links.GetLink(link)[links.Constants.TargetPart];
215
216         /// <summary>
217         /// Возвращает индекс конечной (Target) связи для указанной связи.
218         /// </summary>
219         /// <param name="links">Хранилище связей.</param>
220         /// <param name="link">Связь представленная списком, состоящим из её адреса и
            ↪ содержимого.</param>
221         /// <returns>Индекс конечной связи для указанной связи.</returns>
222         [MethodImpl(MethodImplOptions.AggressiveInlining)]
223         public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
            ↪ link[links.Constants.TargetPart];
224
225         /// <summary>
226         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪ (handler) для каждой подходящей связи.
227         /// </summary>
228         /// <param name="links">Хранилище связей.</param>
229         /// <param name="handler">Обработчик каждой подходящей связи.</param>
230         /// <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>
231         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪ случае.</returns>
232         [MethodImpl(MethodImplOptions.AggressiveInlining)]
233         public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
            ↪ handler, params TLink[] restrictions)
234             => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
                ↪ links.Constants.Continue);
235
236         /// <summary>
237         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪ (handler) для каждой подходящей связи.
238         /// </summary>
239         /// <param name="links">Хранилище связей.</param>
240         /// <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>
241         /// <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>
242         /// <param name="handler">Обработчик каждой подходящей связи.</param>
243         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪ случае.</returns>
244         [MethodImpl(MethodImplOptions.AggressiveInlining)]
245         public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
            ↪ Func<TLink, bool> handler)
246         {
247             var constants = links.Constants;
248             return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
                ↪ constants.Break, constants.Any, source, target);
```

```
249         }
250
251         /// <summary>
252         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪ (handler) для каждой подходящей связи.
253         /// </summary>
254         /// <param name="links">Хранилище связей.</param>
255         /// <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>
256         /// <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>
257         /// <param name="handler">Обработчик каждой подходящей связи.</param>
258         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪ случае.</returns>
259         [MethodImpl(MethodImplOptions.AggressiveInlining)]
260         public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
            ↪ Func<IList<TLink>, TLink> handler)
261         {
262             var constants = links.Constants;
263             return links.Each(handler, constants.Any, source, target);
264         }
265
266         [MethodImpl(MethodImplOptions.AggressiveInlining)]
267         public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
            ↪ restrictions)
268         {
269             long arraySize = (Integer<TLink>)links.Count(restrictions);
270             var array = new IList<TLink>[arraySize];
271             if (arraySize > 0)
272             {
273                 var filler = new ArrayFiller<IList<TLink>, TLink>(array,
                    ↪ links.Constants.Continue);
274                 links.Each(filler.AddAndReturnConstant, restrictions);
275             }
276             return array;
277         }
278
279         [MethodImpl(MethodImplOptions.AggressiveInlining)]
280         public static IList<TLink> AllIndices<TLink>(this ILinks<TLink> links, params TLink[]
            ↪ restrictions)
281         {
282             long arraySize = (Integer<TLink>)links.Count(restrictions);
283             var array = new TLink[arraySize];
284             if (arraySize > 0)
285             {
286                 var filler = new ArrayFiller<TLink, TLink>(array, links.Constants.Continue);
287                 links.Each(filler.AddFirstAndReturnConstant, restrictions);
288             }
289             return array;
290         }
291
292         /// <summary>
293         /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
            ↪ в хранилище связей.
294         /// </summary>
295         /// <param name="links">Хранилище связей.</param>
296         /// <param name="source">Начало связи.</param>
297         /// <param name="target">Конец связи.</param>
298         /// <returns>Значение, определяющее существует ли связь.</returns>
299         [MethodImpl(MethodImplOptions.AggressiveInlining)]
300         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;
301
302         #region Ensure
303         // TODO: May be move to EnsureExtensions or make it both there and here
304
305         [MethodImpl(MethodImplOptions.AggressiveInlining)]
306         public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
            ↪ reference, string argumentName)
307         {
308             if (links.IsInnerReference(reference) && !links.Exists(reference))
309             {
310                 throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
311             }
```

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

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

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

        /// <summary>
        /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="source">Индекс связи, которая является началом для искомой
        ↪  связи.</param>
        /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
        /// <returns>Индекс искомой связи с указанными Source (началом) и Target
        ↪  (концом).</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
        ↪  target)
        {
```

```
            var contants = links.Constants;
            var setter = new Setter<TLink, TLink>(contants.Continue, contants.Break, default);
            links.Each(setter.SetFirstAndReturnFalse, contants.Any, source, target);
            return setter.Result;
        }

        /// <param name="links">Хранилище связей.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
        {
            var link = links.Create();
            return links.Update(link, link, link);
        }

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

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

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

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

        /// <summary>
```

```
517         /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
            ↪ с указанными Source (началом) и Target (концом).
518         /// </summary>
519         /// <param name="links">Хранилище связей.</param>
520         /// <param name="source">Индекс связи, которая является началом на создаваемой
            ↪ связи.</param>
521         /// <param name="target">Индекс связи, которая является концом для создаваемой
            ↪ связи.</param>
522         /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
523         [MethodImpl(MethodImplOptions.AggressiveInlining)]
524         public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
            ↪ target)
525         {
526             var link = links.SearchOrDefault(source, target);
527             if (EqualityComparer<TLink>.Default.Equals(link, default))
528             {
529                 link = links.CreateAndUpdate(source, target);
530             }
531             return link;
532         }
533
534         /// <summary>
535         /// Обновляет связь с указанными началом (Source) и концом (Target)
536         /// на связь с указанными началом (NewSource) и концом (NewTarget).
537         /// </summary>
538         /// <param name="links">Хранилище связей.</param>
539         /// <param name="source">Индекс связи, которая является началом обновляемой
            ↪ связи.</param>
540         /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
541         /// <param name="newSource">Индекс связи, которая является началом связи, на которую
            ↪ выполняется обновление.</param>
542         /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
            ↪ выполняется обновление.</param>
543         /// <returns>Индекс обновлённой связи.</returns>
544         [MethodImpl(MethodImplOptions.AggressiveInlining)]
545         public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
            ↪ TLink target, TLink newSource, TLink newTarget)
546         {
547             var equalityComparer = EqualityComparer<TLink>.Default;
548             var link = links.SearchOrDefault(source, target);
549             if (equalityComparer.Equals(link, default))
550             {
551                 return links.CreateAndUpdate(newSource, newTarget);
552             }
553             if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
                ↪ target))
554             {
555                 return link;
556             }
557             return links.Update(link, newSource, newTarget);
558         }
559
560         /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
561         /// <param name="links">Хранилище связей.</param>
562         /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
563         /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
564         [MethodImpl(MethodImplOptions.AggressiveInlining)]
565         public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
            ↪ target)
566         {
567             var link = links.SearchOrDefault(source, target);
568             if (!EqualityComparer<TLink>.Default.Equals(link, default))
569             {
570                 links.Delete(link);
571                 return link;
572             }
573             return default;
574         }
575
576         /// <summary>Удаляет несколько связей.</summary>
577         /// <param name="links">Хранилище связей.</param>
578         /// <param name="deletedLinks">Список адресов связей к удалению.</param>
579         [MethodImpl(MethodImplOptions.AggressiveInlining)]
580         public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
581         {
582             for (int i = 0; i < deletedLinks.Count; i++)
583             {
```

```
584                 links.Delete(deletedLinks[i]);
585             }
586         }
587 
588         /// <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>
589         public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
590         {
591             var anyConstant = links.Constants.Any;
592             var usagesAsSourceQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
593             links.DeleteByQuery(usagesAsSourceQuery);
594             var usagesAsTargetQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
595             links.DeleteByQuery(usagesAsTargetQuery);
596         }
597 
598         public static void DeleteByQuery<TLink>(this ILinks<TLink> links, Link<TLink> query)
599         {
600             var count = (Integer<TLink>)links.Count(query);
601             if (count > 0)
602             {
603                 var queryResult = new TLink[count];
604                 var queryResultFiller = new ArrayFiller<TLink, TLink>(queryResult,
                    ↪  links.Constants.Continue);
605                 links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
606                 for (var i = (long)count - 1; i >= 0; i--)
607                 {
608                     links.Delete(queryResult[i]);
609                 }
610             }
611         }
612 
613         // TODO: Move to Platform.Data
614         public static bool AreValuesReset<TLink>(this ILinks<TLink> links, TLink linkIndex)
615         {
616             var nullConstant = links.Constants.Null;
617             var equalityComparer = EqualityComparer<TLink>.Default;
618             var link = links.GetLink(linkIndex);
619             for (int i = 1; i < link.Count; i++)
620             {
621                 if (!equalityComparer.Equals(link[i], nullConstant))
622                 {
623                     return false;
624                 }
625             }
626             return true;
627         }
628 
629         // TODO: Create a universal version of this method in Platform.Data (with using of for
            ↪  loop)
630         public static void ResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
631         {
632             var nullConstant = links.Constants.Null;
633             var updateRequest = new Link<TLink>(linkIndex, nullConstant, nullConstant);
634             links.Update(updateRequest);
635         }
636 
637         // TODO: Create a universal version of this method in Platform.Data (with using of for
            ↪  loop)
638         public static void EnforceResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
639         {
640             if (!links.AreValuesReset(linkIndex))
641             {
642                 links.ResetValues(linkIndex);
643             }
644         }
645 
646         /// <summary>
647         /// Merging two usages graphs, all children of old link moved to be children of new link
            ↪  or deleted.
648         /// </summary>
649         public static TLink MergeUsages<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
            ↪  TLink newLinkIndex)
650         {
651             var equalityComparer = EqualityComparer<TLink>.Default;
652             if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
653             {
654                 var constants = links.Constants;
```

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

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

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

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

namespace Platform.Data.Doublets.Incrementers
{
    public class FrequencyIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly TLink _frequencyMarker;
        private readonly TLink _unaryOne;
        private readonly IIncrementer<TLink> _unaryNumberIncrementer;

        public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
        ↪  IIncrementer<TLink> unaryNumberIncrementer)
            : base(links)
        {
            _frequencyMarker = frequencyMarker;
            _unaryOne = unaryOne;
            _unaryNumberIncrementer = unaryNumberIncrementer;
        }

        public TLink Increment(TLink frequency)
        {
            if (_equalityComparer.Equals(frequency, default))
            {
                return Links.GetOrCreate(_unaryOne, _frequencyMarker);
            }
            var source = Links.GetSource(frequency);
            var incrementedSource = _unaryNumberIncrementer.Increment(source);
            return Links.GetOrCreate(incrementedSource, _frequencyMarker);
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Incrementers
{
    public class UnaryNumberIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly TLink _unaryOne;

        public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) : base(links) =>
        ↪  _unaryOne = unaryOne;

        public TLink Increment(TLink unaryNumber)
        {
            if (_equalityComparer.Equals(unaryNumber, _unaryOne))
            {
                return Links.GetOrCreate(_unaryOne, _unaryOne);
            }
            var source = Links.GetSource(unaryNumber);
            var target = Links.GetTarget(unaryNumber);
            if (_equalityComparer.Equals(source, target))
            {
                return Links.GetOrCreate(unaryNumber, _unaryOne);
            }
            else
            {
                return Links.GetOrCreate(source, Increment(target));
            }
        }
    }
}
```

./Platform.Data.Doublets/ISynchronizedLinks.cs

```
using Platform.Data.Constants;

```

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

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

./Platform.Data.Doublets/Link.cs

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

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

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

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

        private const int Length = 3;

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

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

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

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

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

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

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

```

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

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

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

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

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

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

```
using Platform.Interfaces;

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

namespace Platform.Data.Doublets.Numbers.Raw
{
    public class AddressToRawNumberConverter<TLink> : IConverter<TLink>
    {
        public TLink Convert(TLink source) => new Hybrid<TLink>(source, isExternal: true);
    }
}
```

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

```
using Platform.Interfaces;
using Platform.Numbers;

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

namespace Platform.Data.Doublets.Numbers.Raw
{
    public class RawNumberToAddressConverter<TLink> : IConverter<TLink>
```

```
    {
        public TLink Convert(TLink source) => (Integer<TLink>)new
        ↪  Hybrid<TLink>(source).AbsoluteValue;
    }
}
```

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

```
using System.Collections.Generic;
using Platform.Interfaces;
using Platform.Reflection;
using Platform.Numbers;

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

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

        private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;

        public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
        ↪  powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
        ↪  powerOf2ToUnaryNumberConverter;

        public TLink Convert(TLink sourceAddress)
        {
            var number = sourceAddress;
            var nullConstant = Links.Constants.Null;
            var one = Integer<TLink>.One;
            var target = nullConstant;
            for (int i = 0; !_equalityComparer.Equals(number, default) && i <
            ↪  Type<TLink>.BitsLength; i++)
            {
                if (_equalityComparer.Equals(Arithmetic.And(number, one), one))
                {
                    target = _equalityComparer.Equals(target, nullConstant)
                        ? _powerOf2ToUnaryNumberConverter.Convert(i)
                        : Links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
                }
                number = (Integer<TLink>)((ulong)(Integer<TLink>)number >> 1); // Should be
                ↪  Bit.ShiftRight(number, 1)
            }
            return target;
        }
    }
}
```

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

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

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

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

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

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

        public TLink Convert(Doublet<TLink> doublet)
```

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

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

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

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

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

        private readonly TLink[] _unaryNumberPowersOf2;

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

        public TLink Convert(int power)
        {
            Ensure.Always.ArgumentInRange(power, new Range<int>(0, _unaryNumberPowersOf2.Length
            ↪  - 1), nameof(power));
            if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
            {
                return _unaryNumberPowersOf2[power];
            }
            var previousPowerOf2 = Convert(power - 1);
            var powerOf2 = Links.GetOrCreate(previousPowerOf2, previousPowerOf2);
            _unaryNumberPowersOf2[power] = powerOf2;
            return powerOf2;
        }
    }
}
```

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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Numbers;

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

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

        private Dictionary<TLink, TLink> _unaryToUInt64;
        private readonly TLink _unaryOne;

        public UnaryNumberToAddressAddOperationConverter(ILinks<TLink> links, TLink unaryOne)
            : base(links)
        {
```

```
            _unaryOne = unaryOne;
            InitUnaryToUInt64();
        }

        private void InitUnaryToUInt64()
        {
            var one = Integer<TLink>.One;
            _unaryToUInt64 = new Dictionary<TLink, TLink>
            {
                { _unaryOne, one }
            };
            var unary = _unaryOne;
            var number = one;
            for (var i = 1; i < 64; i++)
            {
                unary = Links.GetOrCreate(unary, unary);
                number = Double(number);
                _unaryToUInt64.Add(unary, number);
            }
        }

        public TLink Convert(TLink unaryNumber)
        {
            if (_equalityComparer.Equals(unaryNumber, default))
            {
                return default;
            }
            if (_equalityComparer.Equals(unaryNumber, _unaryOne))
            {
                return Integer<TLink>.One;
            }
            var source = Links.GetSource(unaryNumber);
            var target = Links.GetTarget(unaryNumber);
            if (_equalityComparer.Equals(source, target))
            {
                return _unaryToUInt64[unaryNumber];
            }
            else
            {
                var result = _unaryToUInt64[source];
                TLink lastValue;
                while (!_unaryToUInt64.TryGetValue(target, out lastValue))
                {
                    source = Links.GetSource(target);
                    result = Arithmetic<TLink>.Add(result, _unaryToUInt64[source]);
                    target = Links.GetTarget(target);
                }
                result = Arithmetic<TLink>.Add(result, lastValue);
                return result;
            }
        }

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

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

```
using System.Collections.Generic;
using Platform.Interfaces;
using Platform.Reflection;
using Platform.Numbers;
using System.Runtime.CompilerServices;

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

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

        private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;

        public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
        ↪  TLink> powerOf2ToUnaryNumberConverter)
            : base(links)
```

```
        {
            _unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
            for (int i = 0; i < Type<TLink>.BitsLength; i++)
            {
                _unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
            }
        }

        public TLink Convert(TLink sourceNumber)
        {
            var nullConstant = Links.Constants.Null;
            var source = sourceNumber;
            var target = nullConstant;
            if (!_equalityComparer.Equals(source, nullConstant))
            {
                while (true)
                {
                    if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
                    {
                        SetBit(ref target, powerOf2Index);
                        break;
                    }
                    else
                    {
                        powerOf2Index = _unaryNumberPowerOf2Indicies[Links.GetSource(source)];
                        SetBit(ref target, powerOf2Index);
                        source = Links.GetTarget(source);
                    }
                }
            }
            return target;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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))
    }
}
```

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

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

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

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

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

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

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 =>
            {
                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

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

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

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

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

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

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

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

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

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

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

```
        }

        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))
                {
                    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];
```

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

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

```
                    else //if(source != Any && target != Any)
                    {
                        var link = _sourcesTreeMethods.Search(source, target);
                        return _equalityComparer.Equals(link, Constants.Null) ?
                        ↪  Constants.Continue : handler(GetLinkStruct(link));
                    }
                }
                else
                {
                    if (!Exists(index))
                    {
                        return Constants.Continue;
                    }
                    if (_equalityComparer.Equals(source, Constants.Any) &&
                    ↪  _equalityComparer.Equals(target, Constants.Any))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    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 handler(GetLinkStruct(index));
                        }
                        return Constants.Continue;
                    }
                    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 handler(GetLinkStruct(index));
                    }
                    return Constants.Continue;
                }
            }
            throw new NotSupportedException("Другие размеры и способы ограничений не
            ↪  поддерживаются.");
        }

        /// <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);
```

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

```
            }
        }

        /// <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)
        {
            if (!wasDisposed)
            {
                SetPointers(null);
                _memory.DisposeIfPossible();
            }
        }

        #endregion
    }
}
```

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

```
using System;
using Platform.Unsafe;
using Platform.Collections.Methods.Lists;

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

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    partial class ResizableDirectMemoryLinks<TLink>
    {
        private class UnusedLinksListMethods : CircularDoublyLinkedListMethods<TLink>
        {
            private readonly IntPtr _links;
            private readonly IntPtr _header;

            public UnusedLinksListMethods(IntPtr links, IntPtr header)
            {
                _links = links;
```

```
                _header = header;
            }

            protected override TLink GetFirst() => (_header +
            ↪  LinksHeader.FirstFreeLinkOffset).GetValue<TLink>();

            protected override TLink GetLast() => (_header +
            ↪  LinksHeader.LastFreeLinkOffset).GetValue<TLink>();

            protected override TLink GetPrevious(TLink element) =>
            ↪  (_links.GetElement(LinkSizeInBytes, element) +
            ↪  Link.SourceOffset).GetValue<TLink>();

            protected override TLink GetNext(TLink element) =>
            ↪  (_links.GetElement(LinkSizeInBytes, element) +
            ↪  Link.TargetOffset).GetValue<TLink>();

            protected override TLink GetSize() => (_header +
            ↪  LinksHeader.FreeLinksOffset).GetValue<TLink>();

            protected override void SetFirst(TLink element) => (_header +
            ↪  LinksHeader.FirstFreeLinkOffset).SetValue(element);

            protected override void SetLast(TLink element) => (_header +
            ↪  LinksHeader.LastFreeLinkOffset).SetValue(element);

            protected override void SetPrevious(TLink element, TLink previous) =>
            ↪  (_links.GetElement(LinkSizeInBytes, element) +
            ↪  Link.SourceOffset).SetValue(previous);

            protected override void SetNext(TLink element, TLink next) =>
            ↪  (_links.GetElement(LinkSizeInBytes, element) + Link.TargetOffset).SetValue(next);

            protected override void SetSize(TLink size) => (_header +
            ↪  LinksHeader.FreeLinksOffset).SetValue(size);
        }
    }
}
```

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

```
using System;
using System.Text;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Numbers;
using Platform.Unsafe;
using Platform.Collections.Methods.Trees;
using Platform.Data.Constants;

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

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    partial class ResizableDirectMemoryLinks<TLink>
    {
        private abstract class LinksTreeMethodsBase :
        ↪  SizedAndThreadedAVLBalancedTreeMethods<TLink>
        {
            private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;

            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);

```

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

```
                    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;

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

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

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

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

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

```

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

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

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

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

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

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

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

// ReSharper disable BuiltInTypeReferenceStyle

//#define ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    using id = UInt64;

    public unsafe partial class UInt64ResizableDirectMemoryLinks : DisposableBase, ILinks<id>
    {
        /// <summary>Возвращает размер одной связи в байтах.</summary>
        /// <remarks>
        /// Используется только во вне класса, не рекомедуется использовать внутри.
        /// Так как во вне не обязательно будет доступен unsafe C#.
        /// </remarks>
        public static readonly int LinkSizeInBytes = sizeof(Link);

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

        private struct Link
        {
            public id Source;
            public id Target;
            public id LeftAsSource;
            public id RightAsSource;
            public id SizeAsSource;
            public id LeftAsTarget;
            public id RightAsTarget;
            public id SizeAsTarget;
        }

        private struct LinksHeader
        {
            public id AllocatedLinks;
            public id ReservedLinks;
            public id FreeLinks;
            public id FirstFreeLink;
            public id FirstAsSource;
            public id FirstAsTarget;
            public id LastFreeLink;
            public id Reserved8;
        }

        private readonly long _memoryReservationStep;

        private readonly IResizableDirectMemory _memory;
        private LinksHeader* _header;
        private Link* _links;

        private LinksTargetsTreeMethods _targetsTreeMethods;
        private LinksSourcesTreeMethods _sourcesTreeMethods;

        // 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 id Total => _header->AllocatedLinks - _header->FreeLinks;

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

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

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

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

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

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

```
                    }
                    if (source == Constants.Any && target == Constants.Any)
                    {
                        return handler(GetLinkStruct(index));
                    }
                    var storedLinkValue = GetLinkUnsafe(index);
                    if (source != Constants.Any && target != Constants.Any)
                    {
                        if (storedLinkValue->Source == source &&
                            storedLinkValue->Target == target)
                        {
                            return handler(GetLinkStruct(index));
                        }
                        return Constants.Continue;
                    }
                    var value = default(id);
                    if (source == Constants.Any)
                    {
                        value = target;
                    }
                    if (target == Constants.Any)
                    {
                        value = source;
                    }
                    if (storedLinkValue->Source == value ||
                        storedLinkValue->Target == value)
                    {
                        return handler(GetLinkStruct(index));
                    }
                    return Constants.Continue;
                }
            }
            throw new NotSupportedException("Другие размеры и способы ограничений не
            ↪  поддерживаются.");
        }

        /// <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 id Update(IList<id> 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 (link->Source != Constants.Null)
            {
                _sourcesTreeMethods.Detach(new IntPtr(&_header->FirstAsSource), linkIndex);
            }
            if (link->Target != Constants.Null)
            {
                _targetsTreeMethods.Detach(new IntPtr(&_header->FirstAsTarget), linkIndex);
            }
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
            var leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
            var rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
            if (leftTreeSize != rightTreeSize)
            {
                throw new Exception("One of the trees is broken.");
            }
#endif
            link->Source = values[Constants.SourcePart];
            link->Target = values[Constants.TargetPart];
            if (link->Source != Constants.Null)
            {
                _sourcesTreeMethods.Attach(new IntPtr(&_header->FirstAsSource), linkIndex);
            }
            if (link->Target != Constants.Null)
            {
                _targetsTreeMethods.Attach(new IntPtr(&_header->FirstAsTarget), linkIndex);
            }
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
            leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
            rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
            if (leftTreeSize != rightTreeSize)
            {
```

```
                throw new Exception("One of the trees is broken.");
            }
#endif
            return linkIndex;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private IList<id> GetLinkStruct(id linkIndex)
        {
            var link = GetLinkUnsafe(linkIndex);
            return new UInt64Link(linkIndex, link->Source, link->Target);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private Link* GetLinkUnsafe(id linkIndex) => &_links[linkIndex];

        /// <remarks>
        /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
        ↪  пространство
        /// </remarks>
        public id Create()
        {
            var freeLink = _header->FirstFreeLink;
            if (freeLink != Constants.Null)
            {
                _unusedLinksListMethods.Detach(freeLink);
            }
            else
            {
                if (_header->AllocatedLinks > Constants.MaxPossibleIndex)
                {
                    throw new LinksLimitReachedException(Constants.MaxPossibleIndex);
                }
                if (_header->AllocatedLinks >= _header->ReservedLinks - 1)
                {
                    _memory.ReservedCapacity += _memoryReservationStep;
                    SetPointers(_memory);
                    _header->ReservedLinks = (id)(_memory.ReservedCapacity / sizeof(Link));
                }
                _header->AllocatedLinks++;
                _memory.UsedCapacity += sizeof(Link);
                freeLink = _header->AllocatedLinks;
            }
            return freeLink;
        }

        public void Delete(id link)
        {
            if (link < _header->AllocatedLinks)
            {
                _unusedLinksListMethods.AttachAsFirst(link);
            }
            else if (link == _header->AllocatedLinks)
            {
                _header->AllocatedLinks--;
                _memory.UsedCapacity -= sizeof(Link);
                // 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 (_header->AllocatedLinks > 0 && IsUnusedLink(_header->AllocatedLinks))
                {
                    _unusedLinksListMethods.Detach(_header->AllocatedLinks);
                    _header->AllocatedLinks--;
                    _memory.UsedCapacity -= sizeof(Link);
                }
            }
        }

        /// <remarks>
        /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
        ↪  адрес реально поменялся
        ///
        /// Указатель this.links может быть в том же месте,
        /// так как 0-я связь не используется и имеет такой же размер как Header,
        /// поэтому header размещается в том же месте, что и 0-я связь
        /// </remarks>
        private void SetPointers(IResizableDirectMemory memory)
        {
            if (memory == null)
```

```
            {
                _header = null;
                _links = null;
                _unusedLinksListMethods = null;
                _targetsTreeMethods = null;
                _unusedLinksListMethods = null;
            }
            else
            {
                _header = (LinksHeader*)(void*)memory.Pointer;
                _links = (Link*)(void*)memory.Pointer;
                _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
                _targetsTreeMethods = new LinksTargetsTreeMethods(this);
                _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
            }
        }

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

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

        #region Disposable

        protected override bool AllowMultipleDisposeCalls => true;

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

        #endregion
    }
}
```

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

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

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

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

            public UnusedLinksListMethods(Link* links, LinksHeader* header)
            {
                _links = links;
                _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;

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

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

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

            protected override void SetPrevious(ulong element, ulong previous) =>
            ↪  _links[element].Source = previous;

```

```
            protected override void SetNext(ulong element, ulong next) => _links[element].Target
            ↪  = next;

            protected override void SetSize(ulong size) => _header->FreeLinks = size;
        }
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using System.Text;
using Platform.Collections.Methods.Trees;
using Platform.Data.Constants;

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

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    unsafe partial class UInt64ResizableDirectMemoryLinks
    {
        private abstract class LinksTreeMethodsBase :
        ↪  SizedAndThreadedAVLBalancedTreeMethods<ulong>
        {
            private readonly UInt64ResizableDirectMemoryLinks _memory;
            private readonly LinksCombinedConstants<ulong, ulong, int> _constants;
            protected readonly Link* Links;
            protected readonly LinksHeader* Header;

            protected LinksTreeMethodsBase(UInt64ResizableDirectMemoryLinks memory)
            {
                Links = memory._links;
                Header = memory._header;
                _memory = memory;
                _constants = memory.Constants;
            }

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

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

            public ulong this[ulong index]
            {
                get
                {
                    var root = GetTreeRoot();
                    if (index >= GetSize(root))
                    {
                        return 0;
                    }
                    while (root != 0)
                    {
                        var left = GetLeftOrDefault(root);
                        var leftSize = GetSizeOrZero(left);
                        if (index < leftSize)
                        {
                            root = left;
                            continue;
                        }
                        if (index == leftSize)
                        {
                            return root;
                        }
                        root = GetRightOrDefault(root);
                        index -= leftSize + 1;
                    }
                    return 0; // TODO: Impossible situation exception (only if tree structure
                    ↪  broken)
                }
            }

            // 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;
        }

        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;
            }

            protected override sbyte GetBalance(ulong node)
            {
                var previousValue = Links[node].SizeAsSource;
                //var value = Math.PartialRead(previousValue, 0, 3);
                var value = previousValue & 7;
```

```
223                 var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
                    ↪  124 : value & 3);
224                 return unpackedValue;
225             }
226
227             protected override void SetBalance(ulong node, sbyte value)
228             {
229                 var previousValue = Links[node].SizeAsSource;
230                 var packagedValue = (ulong)((((byte)value >> 5) & 4) | value & 3);
231                 //var modified = Math.PartialWrite(previousValue, packagedValue, 0, 3);
232                 var modified = (previousValue & 4294967288) | (packagedValue & 7);
233                 Links[node].SizeAsSource = modified;
234             }
235
236             protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
237                 => Links[first].Source < Links[second].Source ||
238                   (Links[first].Source == Links[second].Source && Links[first].Target <
                    ↪  Links[second].Target);
239
240             protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
241                 => Links[first].Source > Links[second].Source ||
242                   (Links[first].Source == Links[second].Source && Links[first].Target >
                    ↪  Links[second].Target);
243
244             protected override ulong GetTreeRoot() => Header->FirstAsSource;
245
246             protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
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 ulong Search(ulong source, ulong target)
256             {
257                 var root = Header->FirstAsSource;
258                 while (root != 0)
259                 {
260                     var rootSource = Links[root].Source;
261                     var rootTarget = Links[root].Target;
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                     }
270                     else // node.Key == root.Key
271                     {
272                         return root;
273                     }
274                 }
275                 return 0;
276             }
277
278             [MethodImpl(MethodImplOptions.AggressiveInlining)]
279             private static bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
                ↪  ulong secondSource, ulong secondTarget)
280                 => firstSource < secondSource || (firstSource == secondSource && firstTarget <
                    ↪  secondTarget);
281
282             [MethodImpl(MethodImplOptions.AggressiveInlining)]
283             private static bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
                ↪  ulong secondSource, ulong secondTarget)
284                 => firstSource > secondSource || (firstSource == secondSource && firstTarget >
                    ↪  secondTarget);
285
286             [MethodImpl(MethodImplOptions.AggressiveInlining)]
287             protected override void ClearNode(ulong node)
288             {
```

```
            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)
        {
        }

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

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

```

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

```
                var modified = (previousValue & 4294967287) | ((value ? 1UL : 0UL) << 3);
                Links[node].SizeAsTarget = modified;
            }

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

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

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

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

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

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

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

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

```
using System.Collections.Generic;

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

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

        public override TLink Convert(IList<TLink> sequence)
        {
            var length = sequence.Count;
            if (length < 1)
            {
                return default;
            }
            if (length == 1)
            {
                return sequence[0];
            }
            // Make copy of next layer
            if (length > 2)
            {
                // TODO: Try to use stackalloc (which at the moment is not working with generics) but will be possible with Sigil
                var halvedSequence = new TLink[(length / 2) + (length % 2)];
                HalveSequence(halvedSequence, sequence, length);
                sequence = halvedSequence;
                length = halvedSequence.Length;
```

```
            }
            // Keep creating layer after layer
            while (length > 2)
            {
                HalveSequence(sequence, sequence, length);
                length = (length / 2) + (length % 2);
            }
            return Links.GetOrCreate(sequence[0], sequence[1]);
        }

        private void HalveSequence(IList<TLink> destination, IList<TLink> source, int length)
        {
            var loopedLength = length - (length % 2);
            for (var i = 0; i < loopedLength; i += 2)
            {
                destination[i / 2] = Links.GetOrCreate(source[i], source[i + 1]);
            }
            if (length > loopedLength)
            {
                destination[length / 2] = source[length - 1];
            }
        }
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Collections;
using Platform.Singletons;
using Platform.Numbers;
using Platform.Data.Constants;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;

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

namespace Platform.Data.Doublets.Sequences.Converters
{
    /// <remarks>
    /// TODO: Возможно будет лучше если алгоритм будет выполняться полностью изолированно от Links на этапе сжатия.
    ///     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.
    ///     Как только список/словарь пар был выявлен можно разом выполнить создание всех этих пар, а так же разом выполнить замену.
    /// </remarks>
    public class CompressingConverter<TLink> : LinksListToSequenceConverterBase<TLink>
    {
        private static readonly LinksCombinedConstants<bool, TLink, long> _constants = Default<LinksCombinedConstants<bool, TLink, long>>.Instance;
        private static readonly EqualityComparer<TLink> _equalityComparer = EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        private readonly IConverter<IList<TLink>, TLink> _baseConverter;
        private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
        private readonly TLink _minFrequencyToCompress;
        private readonly bool _doInitialFrequenciesIncrement;
        private Doublet<TLink> _maxDoublet;
        private LinkFrequency<TLink> _maxDoubletData;

        private struct HalfDoublet
        {
            public TLink Element;
            public LinkFrequency<TLink> DoubletData;

            public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
            {
                Element = element;
                DoubletData = doubletData;
            }

            public override string ToString() => $"{Element}: ({DoubletData})";
        }

        public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink> baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
            : 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)
            {
```

```
                var newLength = ReplaceDoublets(copy);
                sequence = new TLink[newLength];
                for (int i = 0; i < newLength; i++)
                {
                    sequence[i] = copy[i].Element;
                }
            }
            return sequence;
        }

        /// <remarks>
        /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding
        /// </remarks>
        private int ReplaceDoublets(HalfDoublet[] copy)
        {
            var oldLength = copy.Length;
            var newLength = copy.Length;
            while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
            {
                var maxDoubletSource = _maxDoublet.Source;
                var maxDoubletTarget = _maxDoublet.Target;
                if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
                {
                    _maxDoubletData.Link = Links.GetOrCreate(maxDoubletSource, maxDoubletTarget);
                }
                var maxDoubletReplacementLink = _maxDoubletData.Link;
                oldLength--;
                var oldLengthMinusTwo = oldLength - 1;
                // Substitute all usages
                int w = 0, r = 0; // (r == read, w == write)
                for (; r < oldLength; r++)
                {
                    if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
                    ↪  _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
                    {
                        if (r > 0)
                        {
                            var previous = copy[w - 1].Element;
                            copy[w - 1].DoubletData.DecrementFrequency();
                            copy[w - 1].DoubletData =
                            ↪  _doubletFrequenciesCache.IncrementFrequency(previous,
                            ↪  maxDoubletReplacementLink);
                        }
                        if (r < oldLengthMinusTwo)
                        {
                            var next = copy[r + 2].Element;
                            copy[r + 1].DoubletData.DecrementFrequency();
                            copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(ma
                            ↪  xDoubletReplacementLink,
                            ↪  next);
                        }
                        copy[w++].Element = maxDoubletReplacementLink;
                        r++;
                        newLength--;
                    }
                    else
                    {
                        copy[w++] = copy[r];
                    }
                }
                if (w < newLength)
                {
                    copy[w] = copy[r];
                }
                oldLength = newLength;
                ResetMaxDoublet();
                UpdateMaxDoublet(copy, newLength);
            }
            return newLength;
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
193         private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
194         {
195             Doublet<TLink> doublet = default;
196             for (var i = 1; i < length; i++)
197             {
198                 doublet.Source = copy[i - 1].Element;
199                 doublet.Target = copy[i].Element;
200                 UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
201             }
202         }
203 
204         [MethodImpl(MethodImplOptions.AggressiveInlining)]
205         private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
206         {
207             var frequency = data.Frequency;
208             var maxFrequency = _maxDoubletData.Frequency;
209             //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
                ↪  (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
                ↪  compression string data (and gives collisions quickly) */ _maxDoublet.Source +
                ↪  _maxDoublet.Target)))
210             if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
211                (_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) */
212             {
213                 _maxDoublet = doublet;
214                 _maxDoubletData = data;
215             }
216         }
217     }
218 }
```

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

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

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

```
1  using System.Collections.Generic;
2  using System.Linq;
3  using Platform.Interfaces;
4 
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6 
7  namespace Platform.Data.Doublets.Sequences.Converters
8  {
9      public class OptimalVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
12         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
13 
14         private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;
15 
16         public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
           ↪  sequenceToItsLocalElementLevelsConverter) : base(links)
17             => _sequenceToItsLocalElementLevelsConverter =
               ↪  sequenceToItsLocalElementLevelsConverter;
18 
19         public override TLink Convert(IList<TLink> sequence)
20         {
21             var length = sequence.Count;
22             if (length == 1)
23             {
```

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

        private static TLink GetGreatestNeigbourLowerThanCurrentOrCurrent(TLink previous, TLink
        ↪ current, TLink next)
        {
            return _comparer.Compare(previous, next) > 0
                ? _comparer.Compare(previous, current) < 0 ? previous : current
                : _comparer.Compare(next, current) < 0 ? next : current;
```

```
        }

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

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

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

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

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

namespace Platform.Data.Doublets.Sequences.Converters
{
    public class SequenceToItsLocalElementLevelsConverter<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<IList<TLink>>
    {
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        private readonly IConverter<Doublet<TLink>, TLink> _linkToItsFrequencyToNumberConveter;

        public SequenceToItsLocalElementLevelsConverter(ILinks<TLink> links,
        ↪  IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter) : base(links)
        ↪  => _linkToItsFrequencyToNumberConveter = linkToItsFrequencyToNumberConveter;

        public IList<TLink> Convert(IList<TLink> sequence)
        {
            var levels = new TLink[sequence.Count];
            levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
            for (var i = 1; i < sequence.Count - 1; i++)
            {
                var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
                var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
                levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
            }
            levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
            ↪  sequence[sequence.Count - 1]);
            return levels;
        }

        public TLink GetFrequencyNumber(TLink source, TLink target) =>
        ↪  _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLink>(source, target));
    }
}
```

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

```
using Platform.Interfaces;

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

namespace Platform.Data.Doublets.Sequences.CreteriaMatchers
{
    public class DefaultSequenceElementCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
    ↪  ICriterionMatcher<TLink>
    {
        public DefaultSequenceElementCriterionMatcher(ILinks<TLink> links) : base(links) { }
        public bool IsMatched(TLink argument) => Links.IsPartialPoint(argument);
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.CreteriaMatchers
{
    public class MarkedSequenceCriterionMatcher<TLink> : ICriterionMatcher<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly ILinks<TLink> _links;
        private readonly TLink _sequenceMarkerLink;
```

```

        public MarkedSequenceCriterionMatcher(ILinks<TLink> links, TLink sequenceMarkerLink)
        {
            _links = links;
            _sequenceMarkerLink = sequenceMarkerLink;
        }

        public bool IsMatched(TLink sequenceCandidate)
            => _equalityComparer.Equals(_links.GetSource(sequenceCandidate), _sequenceMarkerLink)
            || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarkerLink,
            ↪  sequenceCandidate), _links.Constants.Null);
    }
}
```

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

```
using System.Collections.Generic;
using Platform.Collections.Stacks;
using Platform.Data.Doublets.Sequences.HeightProviders;
using Platform.Data.Sequences;

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

namespace Platform.Data.Doublets.Sequences
{
    public class DefaultSequenceAppender<TLink> : LinksOperatorBase<TLink>,
    ↪  ISequenceAppender<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly IStack<TLink> _stack;
        private readonly ISequenceHeightProvider<TLink> _heightProvider;

        public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
        ↪  ISequenceHeightProvider<TLink> heightProvider)
            : base(links)
        {
            _stack = stack;
            _heightProvider = heightProvider;
        }

        public TLink Append(TLink sequence, TLink appendant)
        {
            var cursor = sequence;
            while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
            {
                var source = Links.GetSource(cursor);
                var target = Links.GetTarget(cursor);
                if (_equalityComparer.Equals(_heightProvider.Get(source),
                ↪  _heightProvider.Get(target)))
                {
                    break;
                }
                else
                {
                    _stack.Push(source);
                    cursor = target;
                }
            }
            var left = cursor;
            var right = appendant;
            while (!_equalityComparer.Equals(cursor = _stack.Pop(), Links.Constants.Null))
            {
                right = Links.GetOrCreate(left, right);
                left = cursor;
            }
            return Links.GetOrCreate(left, right);
        }
    }
}
```

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

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

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

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

```
    public class DuplicateSegmentsCounter<TLink> : ICounter<int>
    {
        private readonly IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
        ↪  _duplicateFragmentsProvider;
        public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLink>,
        ↪  IList<TLink>>>> duplicateFragmentsProvider) => _duplicateFragmentsProvider =
        ↪  duplicateFragmentsProvider;
        public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
    }
}
```

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

```
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;

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

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)
                    {
                        WalkAll(sequenceElements);
                    }
                }
                return _links.Constants.Continue;
            });
            var resultList = _groups.ToList();
            var comparer = Default<ItemComparer>.Instance;
            resultList.Sort(comparer);
#if DEBUG
            foreach (var item in resultList)
            {
                PrintDuplicates(item);
            }
#endif
            return resultList;
        }

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

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

        private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
        {
            var duplicates = new List<TLink>();
            var readAsElement = new HashSet<TLink>();
            _sequences.Each(sequence =>
            {
                duplicates.Add(sequence);
                readAsElement.Add(sequence);
                return true; // Continue
            }, segment);
            if (duplicates.Any(x => _visited.Get((Integer<TLink>)x)))
            {
                return new List<TLink>();
            }
            foreach (var duplicate in duplicates)
            {
                var duplicateBitIndex = (long)(Integer<TLink>)duplicate;
                _visited.Set(duplicateBitIndex);
            }
            if (_sequences is Sequences sequencesExperiments)
            {
                var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((H
                ↪  ashSet<ulong>)(object)readAsElement,
                ↪  (IList<ulong>)segment);
                foreach (var partiallyMatchedSequence in partiallyMatched)
                {
                    TLink sequenceIndex = (Integer<TLink>)partiallyMatchedSequence;
                    duplicates.Add(sequenceIndex);
                }
            }
            duplicates.Sort();
            return duplicates;
        }

```

```
        private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
        {
            if (!(_links is ILinks<ulong> ulongLinks))
            {
                return;
            }
            var duplicatesKey = duplicatesItem.Key;
            var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
            Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)})");
            var duplicatesList = duplicatesItem.Value;
            for (int i = 0; i < duplicatesList.Count; i++)
            {
                ulong sequenceIndex = (Integer<TLink>)duplicatesList[i];
                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));
                Console.WriteLine(formatedSequenceStructure);
                var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
                    ulongLinks);
                Console.WriteLine(sequenceString);
            }
            Console.WriteLine();
        }
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Numbers;

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
{
    /// <remarks>
    /// Can be used to operate with many CompressingConverters (to keep global frequencies data
    ///  between them).
    /// TODO: Extract interface to implement frequencies storage inside Links storage
    /// </remarks>
    public class LinkFrequenciesCache<TLink> : LinksOperatorBase<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
            EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        private readonly Dictionary<Doublet<TLink>, LinkFrequency<TLink>> _doubletsCache;
        private readonly ICounter<TLink, TLink> _frequencyCounter;

        public LinkFrequenciesCache(ILinks<TLink> links, ICounter<TLink, TLink> frequencyCounter)
            : base(links)
        {
            _doubletsCache = new Dictionary<Doublet<TLink>, LinkFrequency<TLink>>(4096,
                DoubletComparer<TLink>.Default);
            _frequencyCounter = frequencyCounter;
        }

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

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

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

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

```
                //}
            }
        }
    }
}
```

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

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

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

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

        public LinkFrequency(TLink frequency, TLink link)
        {
            Frequency = frequency;
            Link = link;
        }

        public LinkFrequency() { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void IncrementFrequency() => Frequency = Arithmetic<TLink>.Increment(Frequency);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void DecrementFrequency() => Frequency = Arithmetic<TLink>.Decrement(Frequency);

        public override string ToString() => $"F: {Frequency}, L: {Link}";
    }
}
```

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

```
using Platform.Interfaces;

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
{
    public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLink> :
    ↪ IConverter<Doublet<TLink>, TLink>
    {
        private readonly LinkFrequenciesCache<TLink> _cache;
        public
        ↪ FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink>
        ↪ cache) => _cache = cache;
        public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref source).Frequency;
    }
}
```

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

```
using Platform.Interfaces;

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    public class MarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
    ↪ SequenceSymbolFrequencyOneOffCounter<TLink>
    {
        private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;

        public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
        ↪ ICriterionMatcher<TLink> markedSequenceMatcher, TLink sequenceLink, TLink symbol)
            : base(links, sequenceLink, symbol)
            => _markedSequenceMatcher = markedSequenceMatcher;

        public override TLink Count()
        {
            if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
            {
                return default;
            }
            return base.Count();
```

```
        }
    }
}
```

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

```
using System.Collections.Generic;
using Platform.Interfaces;
using Platform.Numbers;
using Platform.Data.Sequences;

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    public class SequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        protected readonly ILinks<TLink> _links;
        protected readonly TLink _sequenceLink;
        protected readonly TLink _symbol;
        protected TLink _total;

        public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink sequenceLink,
        ↪  TLink symbol)
        {
            _links = links;
            _sequenceLink = sequenceLink;
            _symbol = symbol;
            _total = default;
        }

        public virtual TLink Count()
        {
            if (_comparer.Compare(_total, default) > 0)
            {
                return _total;
            }
            StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
            ↪  IsElement, VisitElement);
            return _total;
        }

        private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
        ↪  _links.IsPartialPoint(x); // TODO: Use SequenceElementCreteriaMatcher instead of
        ↪  IsPartialPoint

        private bool VisitElement(TLink element)
        {
            if (_equalityComparer.Equals(element, _symbol))
            {
                _total = Arithmetic.Increment(_total);
            }
            return true;
        }
    }
}
```

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

```
using Platform.Interfaces;

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    public class TotalMarkedSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
    {
        private readonly ILinks<TLink> _links;
        private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;

        public TotalMarkedSequenceSymbolFrequencyCounter(ILinks<TLink> links,
        ↪  ICriterionMatcher<TLink> markedSequenceMatcher)
        {
            _links = links;
            _markedSequenceMatcher = markedSequenceMatcher;
        }

```

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

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

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

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

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

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

```
1   using System.Collections.Generic;
2   using Platform.Interfaces;
3   using Platform.Numbers;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
8   {
9       public class TotalSequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
10      {
11          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
12          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
13
14          protected readonly ILinks<TLink> _links;
15          protected readonly TLink _symbol;
16          protected readonly HashSet<TLink> _visits;
17          protected TLink _total;
18
19          public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
20          {
21              _links = links;
22              _symbol = symbol;
23              _visits = new HashSet<TLink>();
24              _total = default;
25          }
26
27          public TLink Count()
28          {
```

```
            if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
            {
                return _total;
            }
            CountCore(_symbol);
            return _total;
        }

        private void CountCore(TLink link)
        {
            var any = _links.Constants.Any;
            if (_equalityComparer.Equals(_links.Count(any, link), default))
            {
                CountSequenceSymbolFrequency(link);
            }
            else
            {
                _links.Each(EachElementHandler, any, link);
            }
        }

        protected virtual void CountSequenceSymbolFrequency(TLink link)
        {
            var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
            ↪  link, _symbol);
            _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
        }

        private TLink EachElementHandler(IList<TLink> doublet)
        {
            var constants = _links.Constants;
            var doubletIndex = doublet[constants.IndexPart];
            if (_visits.Add(doubletIndex))
            {
                CountCore(doubletIndex);
            }
            return constants.Continue;
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.HeightProviders
{
    public class CachedSequenceHeightProvider<TLink> : LinksOperatorBase<TLink>,
    ↪  ISequenceHeightProvider<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly TLink _heightPropertyMarker;
        private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
        private readonly IConverter<TLink> _addressToUnaryNumberConverter;
        private readonly IConverter<TLink> _unaryNumberToAddressConverter;
        private readonly IPropertiesOperator<TLink, TLink, TLink> _propertyOperator;

        public CachedSequenceHeightProvider(
            ILinks<TLink> links,
            ISequenceHeightProvider<TLink> baseHeightProvider,
            IConverter<TLink> addressToUnaryNumberConverter,
            IConverter<TLink> unaryNumberToAddressConverter,
            TLink heightPropertyMarker,
            IPropertiesOperator<TLink, TLink, TLink> propertyOperator)
            : base(links)
        {
            _heightPropertyMarker = heightPropertyMarker;
            _baseHeightProvider = baseHeightProvider;
            _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
            _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
            _propertyOperator = propertyOperator;
        }

        public TLink Get(TLink sequence)
        {
            TLink height;
            var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
```

```
            if (_equalityComparer.Equals(heightValue, default))
            {
                height = _baseHeightProvider.Get(sequence);
                heightValue = _addressToUnaryNumberConverter.Convert(height);
                _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
            }
            else
            {
                height = _unaryNumberToAddressConverter.Convert(heightValue);
            }
            return height;
        }
    }
}
```

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

```
using Platform.Interfaces;
using Platform.Numbers;

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

namespace Platform.Data.Doublets.Sequences.HeightProviders
{
    public class DefaultSequenceRightHeightProvider<TLink> : LinksOperatorBase<TLink>,
    ↪ ISequenceHeightProvider<TLink>
    {
        private readonly ICriterionMatcher<TLink> _elementMatcher;

        public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriterionMatcher<TLink>
        ↪ elementMatcher) : base(links) => _elementMatcher = elementMatcher;

        public TLink Get(TLink sequence)
        {
            var height = default(TLink);
            var pairOrElement = sequence;
            while (!_elementMatcher.IsMatched(pairOrElement))
            {
                pairOrElement = Links.GetTarget(pairOrElement);
                height = Arithmetic.Increment(height);
            }
            return height;
        }
    }
}
```

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

```
using Platform.Interfaces;

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

namespace Platform.Data.Doublets.Sequences.HeightProviders
{
    public interface ISequenceHeightProvider<TLink> : IProvider<TLink, TLink>
    {
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.Indexes
{
    public class CachedFrequencyIncrementingSequenceIndex<TLink> : ISequenceIndex<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪ EqualityComparer<TLink>.Default;

        private readonly LinkFrequenciesCache<TLink> _cache;

        public CachedFrequencyIncrementingSequenceIndex(LinkFrequenciesCache<TLink> cache) =>
        ↪ _cache = cache;

        public bool Add(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
```

```
            while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
            { }
            for (; i >= 1; i--)
            {
                _cache.IncrementFrequency(sequence[i - 1], sequence[i]);
            }
            return indexed;
        }

        private bool IsIndexedWithIncrement(TLink source, TLink target)
        {
            var frequency = _cache.GetFrequency(source, target);
            if (frequency == null)
            {
                return false;
            }
            var indexed = !_equalityComparer.Equals(frequency.Frequency, default);
            if (indexed)
            {
                _cache.IncrementFrequency(source, target);
            }
            return indexed;
        }

        public bool MightContain(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
            return indexed;
        }

        private bool IsIndexed(TLink source, TLink target)
        {
            var frequency = _cache.GetFrequency(source, target);
            if (frequency == null)
            {
                return false;
            }
            return !_equalityComparer.Equals(frequency.Frequency, default);
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.Indexes
{
    public class FrequencyIncrementingSequenceIndex<TLink> : SequenceIndex<TLink>,
    ISequenceIndex<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        EqualityComparer<TLink>.Default;

        private readonly IPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
        private readonly IIncrementer<TLink> _frequencyIncrementer;

        public FrequencyIncrementingSequenceIndex(ILinks<TLink> links, IPropertyOperator<TLink,
        TLink> frequencyPropertyOperator, IIncrementer<TLink> frequencyIncrementer)
            : base(links)
        {
            _frequencyPropertyOperator = frequencyPropertyOperator;
            _frequencyIncrementer = frequencyIncrementer;
        }

        public override bool Add(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
            { }
            for (; i >= 1; i--)
            {
                Increment(Links.GetOrCreate(sequence[i - 1], sequence[i]));
            }
            return indexed;
```

```
        }

        private bool IsIndexedWithIncrement(TLink source, TLink target)
        {
            var link = Links.SearchOrDefault(source, target);
            var indexed = !_equalityComparer.Equals(link, default);
            if (indexed)
            {
                Increment(link);
            }
            return indexed;
        }

        private void Increment(TLink link)
        {
            var previousFrequency = _frequencyPropertyOperator.Get(link);
            var frequency = _frequencyIncrementer.Increment(previousFrequency);
            _frequencyPropertyOperator.Set(link, frequency);
        }
    }
}
```

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

```
using System.Collections.Generic;

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

namespace Platform.Data.Doublets.Sequences.Indexes
{
    public interface ISequenceIndex<TLink>
    {
        /// <summary>
        /// Индексирует последовательность глобально, и возвращает значение,
        /// определяющие была ли запрошенная последовательность проиндексирована ранее.
        /// </summary>
        /// <param name="sequence">Последовательность для индексации.</param>
        bool Add(IList<TLink> sequence);

        bool MightContain(IList<TLink> sequence);
    }
}
```

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

```
using System.Collections.Generic;

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

namespace Platform.Data.Doublets.Sequences.Indexes
{
    public class SequenceIndex<TLink> : LinksOperatorBase<TLink>, ISequenceIndex<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

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

        public virtual bool Add(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed =
            ↪  !_equalityComparer.Equals(Links.SearchOrDefault(sequence[i - 1], sequence[i]),
            ↪  default))) { }
            for (; i >= 1; i--)
            {
                Links.GetOrCreate(sequence[i - 1], sequence[i]);
            }
            return indexed;
        }

        public virtual bool MightContain(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed =
            ↪  !_equalityComparer.Equals(Links.SearchOrDefault(sequence[i - 1], sequence[i]),
            ↪  default))) { }
            return indexed;
        }
```

```
    }
}
```

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

```
using System.Collections.Generic;

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

namespace Platform.Data.Doublets.Sequences.Indexes
{
    public class SynchronizedSequenceIndex<TLink> : ISequenceIndex<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly ISynchronizedLinks<TLink> _links;

        public SynchronizedSequenceIndex(ISynchronizedLinks<TLink> links) => _links = links;

        public bool Add(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            var links = _links.Unsync;
            _links.SyncRoot.ExecuteReadOperation(() =>
            {
                while (--i >= 1 && (indexed =
                ↪  !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                ↪  sequence[i]), default))) { }
            });
            if (!indexed)
            {
                _links.SyncRoot.ExecuteWriteOperation(() =>
                {
                    for (; i >= 1; i--)
                    {
                        links.GetOrCreate(sequence[i - 1], sequence[i]);
                    }
                });
            }
            return indexed;
        }

        public bool MightContain(IList<TLink> sequence)
        {
            var links = _links.Unsync;
            return _links.SyncRoot.ExecuteReadOperation(() =>
            {
                var indexed = true;
                var i = sequence.Count;
                while (--i >= 1 && (indexed =
                ↪  !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                ↪  sequence[i]), default))) { }
                return indexed;
            });
        }
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.CompilerServices;
using Platform.Collections;
using Platform.Collections.Lists;
using Platform.Threading.Synchronization;
using Platform.Singletons;
using LinkIndex = System.UInt64;
using Platform.Data.Constants;
using Platform.Data.Sequences;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Collections.Stacks;

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

namespace Platform.Data.Doublets.Sequences
{
    /// <summary>
    /// Представляет коллекцию последовательностей связей.
```

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

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

```
        #endregion

        #region Create

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

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

        #endregion

        #region Each

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

        public bool Each(Func<ulong, bool> handler, IList<ulong> sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                if (sequence.IsNullOrEmpty())
                {
                    return true;
                }
                Links.EnsureEachLinkIsAnyOrExists(sequence);
                if (sequence.Count == 1)
                {
                    var link = sequence[0];
                    if (link == _constants.Any)
                    {
                        return Links.Unsync.Each(_constants.Any, _constants.Any, handler);
                    }
                    return handler(link);
                }
                if (sequence.Count == 2)
                {
                    return Links.Unsync.Each(sequence[0], sequence[1], handler);
```

```
            }
            if (Options.UseIndex && !Options.Index.MightContain(sequence))
            {
                return false;
            }
            return EachCore(handler, sequence);
        });
    }

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

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

    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));
```

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

```
                var newSequenceLink = GetSequenceByElements(newSequenceElements);
                if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
                {
                    if (sequenceLink != _constants.Null)
                    {
                        Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
                    }
                    Links.Unsync.MergeUsages(sequenceElements, newSequenceElements);
                }
                ClearGarbage(sequenceElementsContents.Source);
                ClearGarbage(sequenceElementsContents.Target);
            }
            else
            {
                if (Options.UseSequenceMarker)
                {
                    var sequenceElements = GetSequenceElements(sequence);
                    var sequenceLink = GetSequenceByElements(sequenceElements);
                    var newSequenceElements = GetSequenceElements(newSequence);
                    var newSequenceLink = GetSequenceByElements(newSequenceElements);
                    if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
                    {
                        if (sequenceLink != _constants.Null)
                        {
                            Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
                        }
                        Links.Unsync.MergeUsages(sequenceElements, newSequenceElements);
                    }
                }
                else
                {
                    if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
                    {
                        Links.Unsync.MergeUsages(sequence, newSequence);
                    }
                }
            }
        }

        #endregion

        #region Delete

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

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

```
                var sequenceLink = GetSequenceByElements(sequenceElements);
                if (Options.UseCascadeDelete || CountUsages(link) == 0)
                {
                    if (sequenceLink != _constants.Null)
                    {
                        Links.Unsync.Delete(sequenceLink);
                    }
                    Links.Unsync.Delete(link);
                }
            }
            else
            {
                if (Options.UseCascadeDelete || CountUsages(link) == 0)
                {
                    Links.Unsync.Delete(link);
                }
            }
        }
    }

    #endregion

    #region Compactification

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

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

    #endregion

    #region Garbage Collection

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

```
554                 foreach (var part in Options.Walker.Walk(sequence))
555                 {
556                     if (!handler(part))
557                     {
558                         return false;
559                     }
560                 }
561                 return true;
562             });
563         }
564 
565         public class Matcher : RightSequenceWalker<ulong>
566         {
567             private readonly Sequences _sequences;
568             private readonly IList<LinkIndex> _patternSequence;
569             private readonly HashSet<LinkIndex> _linksInSequence;
570             private readonly HashSet<LinkIndex> _results;
571             private readonly Func<ulong, bool> _stopableHandler;
572             private readonly HashSet<ulong> _readAsElements;
573             private int _filterPosition;
574 
575             public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
                ↪  HashSet<LinkIndex> results, Func<LinkIndex, bool> stopableHandler,
                ↪  HashSet<LinkIndex> readAsElements = null)
576                 : base(sequences.Links.Unsync, new DefaultStack<ulong>())
577             {
578                 _sequences = sequences;
579                 _patternSequence = patternSequence;
580                 _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                    ↪  _constants.Any && x != ZeroOrMany));
581                 _results = results;
582                 _stopableHandler = stopableHandler;
583                 _readAsElements = readAsElements;
584             }
585 
586             protected override bool IsElement(ulong link) => base.IsElement(link) ||
                ↪  (_readAsElements != null && _readAsElements.Contains(link)) ||
                ↪  _linksInSequence.Contains(link);
587 
588             public bool FullMatch(LinkIndex sequenceToMatch)
589             {
590                 _filterPosition = 0;
591                 foreach (var part in Walk(sequenceToMatch))
592                 {
593                     if (!FullMatchCore(part))
594                     {
595                         break;
596                     }
597                 }
598                 return _filterPosition == _patternSequence.Count;
599             }
600 
601             private bool FullMatchCore(LinkIndex element)
602             {
603                 if (_filterPosition == _patternSequence.Count)
604                 {
605                     _filterPosition = -2; // Длиннее чем нужно
606                     return false;
607                 }
608                 if (_patternSequence[_filterPosition] != _constants.Any
609                  && element != _patternSequence[_filterPosition])
610                 {
611                     _filterPosition = -1;
612                     return false; // Начинается/Продолжается иначе
613                 }
614                 _filterPosition++;
615                 return true;
616             }
617 
618             public void AddFullMatchedToResults(ulong sequenceToMatch)
619             {
620                 if (FullMatch(sequenceToMatch))
621                 {
622                     _results.Add(sequenceToMatch);
623                 }
624             }
625 
626             public bool HandleFullMatched(ulong sequenceToMatch)
627             {
628                 if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
```

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

```
                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

```
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;

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

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

        /// <remarks>
        /// 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)Platform.Numbers.Math.Catalan(stopAt - startAt)];
            var last = 0;
            for (var splitter = startAt; splitter < stopAt; splitter++)
            {
                var left = CreateAllVariants2Core(sequence, startAt, splitter);
                var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
                for (var i = 0; i < left.Length; i++)
                {
                    for (var j = 0; j < right.Length; j++)
                    {
                        var variant = Links.Unsync.CreateAndUpdate(left[i], right[j]);
                        if (variant == _constants.Null)
                        {
                            throw new NotImplementedException("Creation cancellation is not implemented.");
                        }
                        variants[last++] = variant;
                    }
                }
            }
            return variants;
        }

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

        private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
        {
            if (sequence.Length == 2)
            {
                var link = Links.Unsync.CreateAndUpdate(sequence[0], sequence[1]);
                if (link == _constants.Null)
                {
                    throw new NotImplementedException("Creation cancellation is not implemented.");
                }
                results.Add(link);
                return results;
            }
            var innerSequenceLength = sequence.Length - 1;
            var innerSequence = new ulong[innerSequenceLength];
            for (var li = 0; li < innerSequenceLength; li++)
            {
                var link = Links.Unsync.CreateAndUpdate(sequence[li], sequence[li + 1]);
                if (link == _constants.Null)
                {
                    throw new NotImplementedException("Creation cancellation is not implemented.");
                }
                for (var isi = 0; isi < li; isi++)
                {
                    innerSequence[isi] = sequence[isi];
                }
                innerSequence[li] = link;
```

```
                for (var isi = li + 1; isi < innerSequenceLength; isi++)
                {
                    innerSequence[isi] = sequence[isi + 1];
                }
                CreateAllVariants1Core(innerSequence, results);
            }
            return results;
        }

        #endregion

        public HashSet<ulong> Each1(params ulong[] sequence)
        {
            var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
            Each1(link =>
            {
                if (!visitedLinks.Contains(link))
                {
                    visitedLinks.Add(link); // изучить почему случаются повторы
                }
                return true;
            }, sequence);
            return visitedLinks;
        }

        private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
        {
            if (sequence.Length == 2)
            {
                Links.Unsync.Each(sequence[0], sequence[1], handler);
            }
            else
            {
                var innerSequenceLength = sequence.Length - 1;
                for (var li = 0; li < innerSequenceLength; li++)
                {
                    var left = sequence[li];
                    var right = sequence[li + 1];
                    if (left == 0 && right == 0)
                    {
                        continue;
                    }
                    var linkIndex = li;
                    ulong[] innerSequence = null;
                    Links.Unsync.Each(left, right, doublet =>
                    {
                        if (innerSequence == null)
                        {
                            innerSequence = new ulong[innerSequenceLength];
                            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;
        }
```

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

```
            }
            return true;
        });
    }

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

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

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

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

    private void TryStepLeftUp(Action<ulong> handler, ulong left, ulong stepFrom)
    {
        var upStep = stepFrom;
        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;
    }

```

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

```
437             });
438         }
439 
440         public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
441         {
442             return _sync.ExecuteReadOperation(() =>
443             {
444                 var results = new HashSet<ulong>();
445                 if (sequence.Length > 0)
446                 {
447                     Links.EnsureEachLinkExists(sequence);
448                     var firstElement = sequence[0];
449                     if (sequence.Length == 1)
450                     {
451                         results.Add(firstElement);
452                         return results;
453                     }
454                     if (sequence.Length == 2)
455                     {
456                         var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
457                         if (doublet != _constants.Null)
458                         {
459                             results.Add(doublet);
460                         }
461                         return results;
462                     }
463                     var matcher = new Matcher(this, sequence, results, null);
464                     if (sequence.Length >= 2)
465                     {
466                         StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
467                     }
468                     var last = sequence.Length - 2;
469                     for (var i = 1; i < last; i++)
470                     {
471                         PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
                            ↪ sequence[i + 1]);
472                     }
473                     if (sequence.Length >= 3)
474                     {
475                         StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
                            ↪ sequence[sequence.Length - 1]);
476                     }
477                 }
478                 return results;
479             });
480         }
481 
482         public const int MaxSequenceFormatSize = 200;
483 
484         public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
            ↪ => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
485 
486         public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
            ↪ elementToString, bool insertComma, params LinkIndex[] knownElements) =>
            ↪ Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
            ↪ elementToString, insertComma, knownElements));
487 
488         private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
            ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
            ↪ LinkIndex[] knownElements)
489         {
490             var linksInSequence = new HashSet<ulong>(knownElements);
491             //var entered = new HashSet<ulong>();
492             var sb = new StringBuilder();
493             sb.Append('{');
494             if (links.Exists(sequenceLink))
495             {
496                 StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
497                     x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
                        ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains
498                     {
499                         if (insertComma && sb.Length > 1)
500                         {
501                             sb.Append(',');
502                         }
503                         //if (entered.Contains(element))
504                         //{
505                         //    sb.Append('{');
```

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

    public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
    ↪  LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
    ↪  Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
    ↪  sequenceLink, elementToString, insertComma, knownElements));

    private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
    ↪  Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
    ↪  LinkIndex[] knownElements)
    {
        var linksInSequence = new HashSet<ulong>(knownElements);
        var entered = new HashSet<ulong>();
        var sb = new StringBuilder();
        sb.Append('{');
        if (links.Exists(sequenceLink))
        {
            StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
                x => linksInSequence.Contains(x) || links.IsFullPoint(x),
                ↪  entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
                {
                    if (insertComma && sb.Length > 1)
                    {
                        sb.Append(',');
                    }
                    if (entered.Contains(element))
                    {
                        sb.Append('{');
                        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);
                }
```

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

```
                {
                    if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
                    {
                        return false;
                    }
                }
                return true;
            }
            return true;
        });
    }

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

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

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

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

    //            firstResults.IntersectWith(lastResults);

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

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

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

    public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
    {
        return _sync.ExecuteReadOperation(() =>
        {
            if (sequence.Length > 0)
            {
                Links.EnsureEachLinkIsAnyOrExists(sequence);
                var firstResults = new HashSet<ulong>();
                var lastResults = new HashSet<ulong>();
                var first = sequence.First(x => x != _constants.Any);
                var last = sequence.Last(x => x != _constants.Any);
                AllUsagesCore(first, firstResults);
                AllUsagesCore(last, lastResults);
                firstResults.IntersectWith(lastResults);
                //for (var i = 0; i < sequence.Length; i++)
                //    AllUsagesCore(sequence[i], results);
                var filteredResults = new HashSet<ulong>();
                var matcher = new Matcher(this, sequence, filteredResults, null);
                matcher.AddAllPartialMatchedToResults(firstResults);
                return filteredResults;
            }
            return new HashSet<ulong>();
        });
    }

    public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
    ↪  IList<ulong> sequence)
    {
        return _sync.ExecuteReadOperation(() =>
        {
            if (sequence.Count > 0)
            {
                Links.EnsureEachLinkExists(sequence);
                var results = new HashSet<LinkIndex>();
                //var nextResults = new HashSet<ulong>();
```

```
                    //for (var i = 0; i < sequence.Length; i++)
                    //{
                    //    AllUsagesCore(sequence[i], nextResults);
                    //    if (results.IsNullOrEmpty())
                    //    {
                    //        results = nextResults;
                    //        nextResults = new HashSet<ulong>();
                    //    }
                    //    else
                    //    {
                    //        results.IntersectWith(nextResults);
                    //        nextResults.Clear();
                    //    }
                    //}
                    var collector1 = new AllUsagesCollector1(Links.Unsync, results);
                    collector1.Collect(Links.Unsync.GetLink(sequence[0]));
                    var next = new HashSet<ulong>();
                    for (var i = 1; i < sequence.Count; i++)
                    {
                        var collector = new AllUsagesCollector1(Links.Unsync, next);
                        collector.Collect(Links.Unsync.GetLink(sequence[i]));

                        results.IntersectWith(next);
                        next.Clear();
                    }
                    var filteredResults = new HashSet<ulong>();
                    var matcher = new Matcher(this, sequence, filteredResults, null,
                    ↪  readAsElements);
                    matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
                    ↪  x)); // OrderBy is a Hack
                    return filteredResults;
                }
                return new HashSet<ulong>();
            });
        }

        // Does not work
        public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
        ↪  params ulong[] sequence)
        {
            var visited = new HashSet<ulong>();
            var results = new HashSet<ulong>();
            var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
            ↪  true; }, readAsElements);
            var last = sequence.Length - 1;
            for (var i = 0; i < last; i++)
            {
                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)
                    //////{
                    //////    var usages = new HashSet<ulong>();
                    //////    for (int i = 0; i < sequence.Length; i++)
                    //////    {
                    //////        AllUsagesCore(sequence[i], usages);
                    //////    }
                    //////    //for (int i = 0; i < matches[0].Count; i++)
                    //////    //    AllUsagesCore(matches[0][i], usages);
                    //////    //usages.UnionWith(matches[0]);
                    //////    return usages.ToList();
                    //////}
                    var firstLinkUsages = new HashSet<ulong>();
                    AllUsagesCore(sequence[0], firstLinkUsages);
                    firstLinkUsages.Add(sequence[0]);
                    //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
                    ↪  sequence[0] }; // or all sequences, containing this element?
                    //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
                    ↪  1).ToList();
                    var results = new HashSet<ulong>();
                    foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
                    ↪  firstLinkUsages, 1))
                    {
                        AllUsagesCore(match, results);
                    }
                    return results.ToList();
                }
                return new List<ulong>();
            });
        }

        /// <remarks>
        /// TODO: Может потробоваться ограничение на уровень глубины рекурсии
```

```
        /// </remarks>
        public HashSet<ulong> AllUsages(ulong link)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var usages = new HashSet<ulong>();
                AllUsagesCore(link, usages);
                return usages;
            });
        }

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

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

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

```

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

```
            _totals = totals;
        }

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

        private bool IsElement(ulong link)
        {
            //_linksInSequence.Contains(link) ||
            return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
            ↪  link;
        }

        private bool CalculateCore(ulong link)
        {
            // TODO: Проработать защиту от зацикливания
            // Основано на SequenceWalker.WalkLeft
            Func<ulong, ulong> getSource = _links.Unsync.GetSource;
            Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
            Func<ulong, bool> isElement = IsElement;
            void visitLeaf(ulong parent)
            {
                if (link != parent)
                {
                    _totals[parent]++;
                }
            }
            void visitNode(ulong parent)
            {
                if (link != parent)
                {
                    _totals[parent]++;
                }
            }
            var stack = new Stack();
            var element = link;
            if (isElement(element))
            {
                visitLeaf(element);
            }
            else
            {
                while (true)
                {
                    if (isElement(element))
                    {
                        if (stack.Count == 0)
                        {
                            break;
                        }
                        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
    {
```

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

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

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

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

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

    public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
    {
        return _sync.ExecuteReadOperation(() =>
        {
            var results = new BitString((long)Links.Unsync.Count() + 1); // new
            ↪  BitArray((int)_links.Total + 1);
            if (linksToConnect.Length > 0)
            {
                Links.EnsureEachLinkExists(linksToConnect);
                var collector1 = new AllUsagesCollector2(Links.Unsync, results);
                collector1.Collect(linksToConnect[0]);
                for (var i = 1; i < linksToConnect.Length; i++)
                {
                    var next = new BitString((long)Links.Unsync.Count() + 1); //new
                    ↪  BitArray((int)_links.Total + 1);
                    var collector = new AllUsagesCollector2(Links.Unsync, next);
                    collector.Collect(linksToConnect[i]);
                    results = results.And(next);
                }
            }
            return results.GetSetUInt64Indices();
        });
    }

    private static ulong[] Simplify(ulong[] sequence)
    {
        // Считаем новый размер последовательности
        long newLength = 0;
        var zeroOrManyStepped = false;
        for (var i = 0; i < sequence.Length; i++)
        {
            if (sequence[i] == ZeroOrMany)
            {
                if (zeroOrManyStepped)
```

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

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

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

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

    #region From Triplets

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

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

```
        }

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

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

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

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

        public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
        {
            var result = new ulong[5];
            TryStepLeft(startLink, leftLink, result, 0);
            Links.Each(startLink, _constants.Any, couple =>
            {
                if (couple != startLink)
                {
                    if (TryStepLeft(couple, leftLink, result, 2))
                    {
                        return false;
                    }
                }
                return true;
            });
            if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
            {
                result[4] = leftLink;
            }
            return result;
        }

        public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
        {
            var added = 0;
            Links.Each(_constants.Any, startLink, couple =>
            {
                if (couple != startLink)
                {
                    var coupleSource = Links.GetSource(couple);
                    if (coupleSource == leftLink)
                    {
                        result[offset] = couple;
                        if (++added == 2)
                        {
                            return false;
                        }
                    }
```

```
                    else if (Links.GetTarget(coupleSource) == leftLink) // coupleSource.Linker
                    ↪  == Net.And &&
                    {
                        result[offset + 1] = couple;
                        if (++added == 2)
                        {
                            return false;
                        }
                    }
                }
                return true;
            });
            return added > 0;
        }

        #endregion

        #region Walkers

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

            #region Pattern Match

            enum PatternBlockType
            {
                Undefined,
                Gap,
                Elements
            }

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

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

            #endregion

            public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
            ↪  HashSet<LinkIndex> results)
                : base(sequences.Links.Unsync, new DefaultStack<ulong>())
            {
                _sequences = sequences;
                _patternSequence = patternSequence;
                _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                ↪  _constants.Any && x != ZeroOrMany));
                _results = results;
                _pattern = CreateDetailedPattern();
            }

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

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

            private List<PatternBlock> CreateDetailedPattern()
            {
                var pattern = new List<PatternBlock>();
```

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

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

```
                    _patternPosition++;
                    _sequencePosition = 0;
                    return false; // Полное соответствие
                }
                else
                {
                    if (_sequencePosition > currentPatternBlock.Stop)
                    {
                        return false; // Соответствие невозможно
                    }
                    var nextPatternBlock = _pattern[_patternPosition + 1];
                    if (_patternSequence[nextPatternBlock.Start] == element)
                    {
                        if (nextPatternBlock.Start < nextPatternBlock.Stop)
                        {
                            _patternPosition++;
                            _sequencePosition = 1;
                        }
                        else
                        {
                            _patternPosition += 2;
                            _sequencePosition = 0;
                        }
                    }
                }
            }
            else // currentPatternBlock.Type == PatternBlockType.Elements
            {
                var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
                if (_patternSequence[patternElementPosition] != element)
                {
                    return false; // Соответствие невозможно
                }
                if (patternElementPosition == currentPatternBlock.Stop)
                {
                    _patternPosition++;
                    _sequencePosition = 0;
                }
                else
                {
                    _sequencePosition++;
                }
            }
            return true;
            //if (_patternSequence[_patternPosition] != element)
            //    return false;
            //else
            //{
            //    _sequencePosition++;
            //    _patternPosition++;
            //    return true;
            //}
            ////////
            //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

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

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

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

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

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

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

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

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

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

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

```
        }

        public void ValidateOptions()
        {
            if (UseGarbageCollection && !UseSequenceMarker)
            {
                throw new NotSupportedException("To use garbage collection UseSequenceMarker option must be on.");
            }
        }
    }
}
```

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

```
using System.Collections.Generic;

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

namespace Platform.Data.Doublets.Sequences.Walkers
{
    public interface ISequenceWalker<TLink>
    {
        IEnumerable<TLink> Walk(TLink sequence);
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections.Stacks;

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

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

        public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links, stack, links.IsPartialPoint) { }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override IEnumerable<TLink> WalkContents(TLink element)
        {
            var parts = Links.GetLink(element);
            var start = Links.Constants.IndexPart + 1;
            for (var i = parts.Count - 1; i >= start; i--)
            {
                var part = parts[i];
                if (IsElement(part))
                {
                    yield return part;
                }
            }
        }
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

//#define USEARRAYPOOL
#if USEARRAYPOOL
using Platform.Collections;
```

```
#endif

namespace Platform.Data.Doublets.Sequences.Walkers
{
    public class LeveledSequenceWalker<TLink> : LinksOperatorBase<TLink>, ISequenceWalker<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly Func<TLink, bool> _isElement;

        public LeveledSequenceWalker(ILinks<TLink> links, Func<TLink, bool> isElement) :
        ↪  base(links) => _isElement = isElement;

        public LeveledSequenceWalker(ILinks<TLink> links) : base(links) => _isElement =
        ↪  Links.IsPartialPoint;

        public IEnumerable<TLink> Walk(TLink sequence) => ToArray(sequence);

        public TLink[] ToArray(TLink sequence)
        {
            var length = 1;
            var array = new TLink[length];
            array[0] = sequence;
            if (_isElement(sequence))
            {
                return array;
            }
            bool hasElements;
            do
            {
                length *= 2;
#if USEARRAYPOOL
                var nextArray = ArrayPool.Allocate<ulong>(length);
#else
                var nextArray = new TLink[length];
#endif
                hasElements = false;
                for (var i = 0; i < array.Length; i++)
                {
                    var candidate = array[i];
                    if (_equalityComparer.Equals(array[i], default))
                    {
                        continue;
                    }
                    var doubletOffset = i * 2;
                    if (_isElement(candidate))
                    {
                        nextArray[doubletOffset] = candidate;
                    }
                    else
                    {
                        var link = Links.GetLink(candidate);
                        var linkSource = Links.GetSource(link);
                        var linkTarget = Links.GetTarget(link);
                        nextArray[doubletOffset] = linkSource;
                        nextArray[doubletOffset + 1] = linkTarget;
                        if (!hasElements)
                        {
                            hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
                        }
                    }
                }
#if USEARRAYPOOL
                if (array.Length > 1)
                {
                    ArrayPool.Free(array);
                }
#endif
                array = nextArray;
            }
            while (hasElements);
            var filledElementsCount = CountFilledElements(array);
            if (filledElementsCount == array.Length)
            {
                return array;
            }
            else
            {
                return CopyFilledElements(array, filledElementsCount);
```

```
            }
        }

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

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

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

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections.Stacks;

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

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

        public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links,
        ↪  stack, links.IsPartialPoint) { }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override IEnumerable<TLink> WalkContents(TLink element)
        {
            var parts = Links.GetLink(element);
            for (var i = Links.Constants.IndexPart + 1; i < parts.Count; i++)
            {
                var part = parts[i];
                if (IsElement(part))
                {
                    yield return part;
                }
            }
        }
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections.Stacks;

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

namespace Platform.Data.Doublets.Sequences.Walkers
{
    public abstract class SequenceWalkerBase<TLink> : LinksOperatorBase<TLink>,
    ↪ ISequenceWalker<TLink>
    {
        private readonly IStack<TLink> _stack;
        private readonly Func<TLink, bool> _isElement;

        protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
        ↪ isElement) : base(links)
        {
            _stack = stack;
            _isElement = isElement;
        }

        protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : this(links,
        ↪ stack, links.IsPartialPoint)
        {
        }

        public IEnumerable<TLink> Walk(TLink sequence)
        {
            _stack.Clear();
            var element = sequence;
            if (IsElement(element))
            {
                yield return element;
            }
            else
            {
                while (true)
                {
                    if (IsElement(element))
                    {
                        if (_stack.IsEmpty)
                        {
                            break;
                        }
                        element = _stack.Pop();
                        foreach (var output in WalkContents(element))
                        {
                            yield return output;
                        }
                        element = GetNextElementAfterPop(element);
                    }
                    else
                    {
                        _stack.Push(element);
                        element = GetNextElementAfterPush(element);
                    }
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool IsElement(TLink elementLink) => _isElement(elementLink);

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract IEnumerable<TLink> WalkContents(TLink element);
    }
}
```

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

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

```

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

namespace Platform.Data.Doublets.Stacks
{
    public class Stack<TLink> : IStack<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly ILinks<TLink> _links;
        private readonly TLink _stack;

        public bool IsEmpty => _equalityComparer.Equals(Peek(), _stack);

        public Stack(ILinks<TLink> links, TLink stack)
        {
            _links = links;
            _stack = stack;
        }

        private TLink GetStackMarker() => _links.GetSource(_stack);

        private TLink GetTop() => _links.GetTarget(_stack);

        public TLink Peek() => _links.GetTarget(GetTop());

        public TLink Pop()
        {
            var element = Peek();
            if (!_equalityComparer.Equals(element, _stack))
            {
                var top = GetTop();
                var previousTop = _links.GetSource(top);
                _links.Update(_stack, GetStackMarker(), previousTop);
                _links.Delete(top);
            }
            return element;
        }

        public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
        ↪  _links.GetOrCreate(GetTop(), element));
    }
}
```

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

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

namespace Platform.Data.Doublets.Stacks
{
    public static class StackExtensions
    {
        public static TLink CreateStack<TLink>(this ILinks<TLink> links, TLink stackMarker)
        {
            var stackPoint = links.CreatePoint();
            var stack = links.Update(stackPoint, stackMarker, stackPoint);
            return stack;
        }
    }
}
```

./Platform.Data.Doublets/SynchronizedLinks.cs

```
using System;
using System.Collections.Generic;
using Platform.Data.Constants;
using Platform.Data.Doublets;
using Platform.Threading.Synchronization;

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

namespace Platform.Data.Doublets
{
    /// <remarks>
    /// TODO: Autogeneration of synchronized wrapper (decorator).
    /// TODO: Try to unfold code of each method using IL generation for performance improvements.
    /// TODO: Or even to unfold multiple layers of implementations.
    /// </remarks>
    public class SynchronizedLinks<T> : ISynchronizedLinks<T>
    {
        public LinksCombinedConstants<T, T, int> Constants { get; }
        public ISynchronization SyncRoot { get; }
```

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

./Platform.Data.Doublets/UInt64Link.cs

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

```
            Index = values.Count > _constants.IndexPart ? values[_constants.IndexPart] :
            ↪ _constants.Null;
            Source = values.Count > _constants.SourcePart ? values[_constants.SourcePart] :
            ↪ _constants.Null;
            Target = values.Count > _constants.TargetPart ? values[_constants.TargetPart] :
            ↪ _constants.Null;
        }

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

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

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

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

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

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

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

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

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

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

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

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

        #region IList

        public ulong this[int index]
        {
            get
            {
                Ensure.OnDebug.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;
```

```
108
109         IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
110
111         public IEnumerator<ulong> GetEnumerator()
112         {
113             yield return Index;
114             yield return Source;
115             yield return Target;
116         }
117
118         public void Add(ulong item) => throw new NotSupportedException();
119
120         public void Clear() => throw new NotSupportedException();
121
122         public bool Contains(ulong item) => IndexOf(item) >= 0;
123
124         public void CopyTo(ulong[] array, int arrayIndex)
125         {
126             Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
127             Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
                ↪  nameof(arrayIndex));
128             if (arrayIndex + Length > array.Length)
129             {
130                 throw new ArgumentException();
131             }
132             array[arrayIndex++] = Index;
133             array[arrayIndex++] = Source;
134             array[arrayIndex] = Target;
135         }
136
137         public bool Remove(ulong item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
138
139         public int IndexOf(ulong item)
140         {
141             if (Index == item)
142             {
143                 return _constants.IndexPart;
144             }
145             if (Source == item)
146             {
147                 return _constants.SourcePart;
148             }
149             if (Target == item)
150             {
151                 return _constants.TargetPart;
152             }
153
154             return -1;
155         }
156
157         public void Insert(int index, ulong item) => throw new NotSupportedException();
158
159         public void RemoveAt(int index) => throw new NotSupportedException();
160
161         #endregion
162     }
163 }
```

./Platform.Data.Doublets/UInt64LinkExtensions.cs

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

./Platform.Data.Doublets/UInt64LinksExtensions.cs

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

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

namespace Platform.Data.Doublets
{
    public static class UInt64LinksExtensions
    {
        public static readonly LinksCombinedConstants<bool, ulong, int> Constants =
        ↪  Default<LinksCombinedConstants<bool, ulong, int>>.Instance;

        public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);

        public static void EnsureEachLinkExists(this ILinks<ulong> links, IList<ulong> sequence)
        {
            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>();
```

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

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

./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs

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

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

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

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

```
139         ///     3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
            ↪  операции (трансформации)
140         ///        будут записаны в лог.
141         ///
142         /// </remarks>
143         public class Transaction : DisposableBase
144         {
145             private readonly Queue<Transition> _transitions;
146             private readonly UInt64LinksTransactionsLayer _layer;
147             public bool IsCommitted { get; private set; }
148             public bool IsReverted { get; private set; }
149
150             public Transaction(UInt64LinksTransactionsLayer layer)
151             {
152                 _layer = layer;
153                 if (_layer._currentTransactionId != 0)
154                 {
155                     throw new NotSupportedException("Nested transactions not supported.");
156                 }
157                 IsCommitted = false;
158                 IsReverted = false;
159                 _transitions = new Queue<Transition>();
160                 SetCurrentTransaction(layer, this);
161             }
162
163             public void Commit()
164             {
165                 EnsureTransactionAllowsWriteOperations(this);
166                 while (_transitions.Count > 0)
167                 {
168                     var transition = _transitions.Dequeue();
169                     _layer._transitions.Enqueue(transition);
170                 }
171                 _layer._lastCommitedTransactionId = _layer._currentTransactionId;
172                 IsCommitted = true;
173             }
174
175             private void Revert()
176             {
177                 EnsureTransactionAllowsWriteOperations(this);
178                 var transitionsToRevert = new Transition[_transitions.Count];
179                 _transitions.CopyTo(transitionsToRevert, 0);
180                 for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
181                 {
182                     _layer.RevertTransition(transitionsToRevert[i]);
183                 }
184                 IsReverted = true;
185             }
186
187             public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
                ↪  Transaction transaction)
188             {
189                 layer._currentTransactionId = layer._lastCommitedTransactionId + 1;
190                 layer._currentTransactionTransitions = transaction._transitions;
191                 layer._currentTransaction = transaction;
192             }
193
194             public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
195             {
196                 if (transaction.IsReverted)
197                 {
198                     throw new InvalidOperationException("Transation is reverted.");
199                 }
200                 if (transaction.IsCommitted)
201                 {
202                     throw new InvalidOperationException("Transation is commited.");
203                 }
204             }
205
206             protected override void Dispose(bool manual, bool wasDisposed)
207             {
208                 if (!wasDisposed && _layer != null && !_layer.IsDisposed)
209                 {
210                     if (!IsCommitted && !IsReverted)
211                     {
212                         Revert();
213                     }
214                     _layer.ResetCurrentTransation();
215                 }
```

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

```
        }

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

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

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

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

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

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

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

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

        private void DisposeTransitions()
        {
            try
            {
                var pusher = _transitionsPusher;
                if (pusher != null)
                {
                    _transitionsPusher = null;
                    pusher.Wait();
                }
                if (_transitions != null)
                {
                    PushTransitions();
```

```
                }
                _log.DisposeIfPossible();
                FileHelpers.WriteFirst(_logAddress, _lastCommitedTransition);
            }
            catch
            {
            }
        }

        #region DisposalBase

        protected override void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                DisposeTransitions();
            }
            base.Dispose(manual, wasDisposed);
        }

        #endregion
    }
}
```

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

```
using Platform.Interfaces;
using Platform.Numbers;

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

namespace Platform.Data.Doublets.Unicode
{
    public class CharToUnicodeSymbolConverter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<char, TLink>
    {
        private readonly IConverter<TLink> _addressToNumberConverter;
        private readonly TLink _unicodeSymbolMarker;

        public CharToUnicodeSymbolConverter(ILinks<TLink> links, IConverter<TLink>
        ↪ addressToNumberConverter, TLink unicodeSymbolMarker) : base(links)
        {
            _addressToNumberConverter = addressToNumberConverter;
            _unicodeSymbolMarker = unicodeSymbolMarker;
        }

        public TLink Convert(char source)
        {
            var unaryNumber = _addressToNumberConverter.Convert((Integer<TLink>)source);
            return Links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
        }
    }
}
```

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

```
using Platform.Data.Doublets.Sequences.Indexes;
using Platform.Interfaces;
using System.Collections.Generic;

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

namespace Platform.Data.Doublets.Unicode
{
    public class StringToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<string, TLink>
    {
        private readonly IConverter<char, TLink> _charToUnicodeSymbolConverter;
        private readonly ISequenceIndex<TLink> _index;
        private readonly IConverter<IList<TLink>, TLink> _listToSequenceLinkConverter;
        private readonly TLink _unicodeSequenceMarker;

        public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
        ↪ charToUnicodeSymbolConverter, ISequenceIndex<TLink> index, IConverter<IList<TLink>,
        ↪ TLink> listToSequenceLinkConverter, TLink unicodeSequenceMarker) : base(links)
        {
            _charToUnicodeSymbolConverter = charToUnicodeSymbolConverter;
            _index = index;
            _listToSequenceLinkConverter = listToSequenceLinkConverter;
            _unicodeSequenceMarker = unicodeSequenceMarker;
        }

```

```
        public TLink Convert(string source)
        {
            var elements = new List<TLink>();
            for (int i = 0; i < source.Length; i++)
            {
                elements.Add(_charToUnicodeSymbolConverter.Convert(source[i]));
            }
            _index.Add(elements);
            var sequence = _listToSequenceLinkConverter.Convert(elements);
            return Links.GetOrCreate(sequence, _unicodeSequenceMarker);
        }
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.Runtime.CompilerServices;
using System.Text;
using Platform.Data.Sequences;

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

namespace Platform.Data.Doublets.Unicode
{
    public class UnicodeMap
    {
        public static readonly ulong FirstCharLink = 1;
        public static readonly ulong LastCharLink = FirstCharLink + char.MaxValue;
        public static readonly ulong MapSize = 1 + char.MaxValue;

        private readonly ILinks<ulong> _links;
        private bool _initialized;

        public UnicodeMap(ILinks<ulong> links) => _links = links;

        public static UnicodeMap InitNew(ILinks<ulong> links)
        {
            var map = new UnicodeMap(links);
            map.Init();
            return map;
        }

        public void Init()
        {
            if (_initialized)
            {
                return;
            }
            _initialized = true;
            var firstLink = _links.CreatePoint();
            if (firstLink != FirstCharLink)
            {
                _links.Delete(firstLink);
            }
            else
            {
                for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
                {
                    // From NIL to It (NIL -> Character) transformation meaning, (or infinite
                    ↪  amount of NIL characters before actual Character)
                    var createdLink = _links.CreatePoint();
                    _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;
```

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

```
                    innerSequence[i - offset] = FromCharToLink(sequence[i]);
                }
                result.Add(innerSequence);
                offset += relativeLength;
            }
            return result;
        }

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

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

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

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

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

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

```
using System;
using System.Linq;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Interfaces;
```

```

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

namespace Platform.Data.Doublets.Unicode
{
    public class UnicodeSequenceToStringConverter<TLink> : LinksOperatorBase<TLink>, IConverter<TLink, string>
    {
        private readonly ICriterionMatcher<TLink> _unicodeSequenceCriterionMatcher;
        private readonly ISequenceWalker<TLink> _sequenceWalker;
        private readonly IConverter<TLink, char> _unicodeSymbolToCharConverter;

        public UnicodeSequenceToStringConverter(ILinks<TLink> links, ICriterionMatcher<TLink> unicodeSequenceCriterionMatcher, ISequenceWalker<TLink> sequenceWalker, IConverter<TLink, char> unicodeSymbolToCharConverter) : base(links)
        {
            _unicodeSequenceCriterionMatcher = unicodeSequenceCriterionMatcher;
            _sequenceWalker = sequenceWalker;
            _unicodeSymbolToCharConverter = unicodeSymbolToCharConverter;
        }

        public string Convert(TLink source)
        {
            if(!_unicodeSequenceCriterionMatcher.IsMatched(source))
            {
                throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is not a unicode sequence.");
            }
            var sequence = Links.GetSource(source);
            var charArray = _sequenceWalker.Walk(sequence).Select(_unicodeSymbolToCharConverter.Convert).ToArray();
            return new string(charArray);
        }
    }
}
```

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

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

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

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

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

```
using System;
using Platform.Interfaces;
using Platform.Numbers;

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

namespace Platform.Data.Doublets.Unicode
{
    public class UnicodeSymbolToCharConverter<TLink> : LinksOperatorBase<TLink>, IConverter<TLink, char>
    {
        private readonly IConverter<TLink> _numberToAddressConverter;
        private readonly ICriterionMatcher<TLink> _unicodeSymbolCriterionMatcher;

        public UnicodeSymbolToCharConverter(ILinks<TLink> links, IConverter<TLink> numberToAddressConverter, ICriterionMatcher<TLink> unicodeSymbolCriterionMatcher) : base(links)
        {
            _numberToAddressConverter = numberToAddressConverter;
            _unicodeSymbolCriterionMatcher = unicodeSymbolCriterionMatcher;
        }
```

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

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

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

```
        }
    }
}
```

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

```
using System;
using System.Collections.Generic;
using Xunit;
using Platform.Diagnostics;

namespace Platform.Data.Doublets.Tests
{
    public static class EqualityTests
    {
        protected class UInt64EqualityComparer : IEqualityComparer<ulong>
        {
            public bool Equals(ulong x, ulong y) => x == y;

            public int GetHashCode(ulong obj) => obj.GetHashCode();
        }

        private static bool Equals1<T>(T x, T y) => Equals(x, y);

        private static bool Equals2<T>(T x, T y) => x.Equals(y);

        private static bool Equals3(ulong x, ulong y) => x == y;

        [Fact]
        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);
                }
            });
```

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

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

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

```
        private static void Using<TLink>(Action<ILinks<TLink>> action)
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪ ResizableDirectMemoryLinks<TLink>>>())
            {
                action(scope.Use<ILinks<TLink>>());
            }
        }
    }
}
```

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

```
using System;
using System.Linq;
using System.Collections.Generic;
using Xunit;
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.PropertyOperators;
using Platform.Data.Doublets.Incrementers;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Sequences.Indexes;
using Platform.Data.Doublets.Unicode;
using Platform.Data.Doublets.Numbers.Unary;

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

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

                links.UseUnicode();

                var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);

                var meaningRoot = links.CreatePoint();
                var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
                var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
                var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
                ↪ constants.Itself);

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

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

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

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

```
        {
            using (var scope = new TempLinksTestScope(useSequences: false))
            {
                var links = scope.Links;

                links.UseUnicode();

                var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);

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

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

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

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

                var sequenceToItsLocalElementLevelsConverter = new
                ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                ↪  linkToItsFrequencyNumberConverter);
                var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                ↪  sequenceToItsLocalElementLevelsConverter);

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

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

        private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
        ↪  SequenceToItsLocalElementLevelsConverter<ulong>
        ↪  sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
        ↪  OptimalVariantConverter<ulong> optimalVariantConverter)
        {
            index.Add(sequence);

            var optimalVariant = optimalVariantConverter.Convert(sequence);

            var readSequence1 = sequences.ToList(optimalVariant);

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

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

```
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];
                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 readSequence1 = sequences.ToList(balancedVariant); sw2.Stop();

                var sw3 = Stopwatch.StartNew();
                var readSequence2 = new List<ulong>();
                SequenceWalker.WalkRight(balancedVariant,
                                         links.GetSource,
                                         links.GetTarget,
                                         links.IsPartialPoint,
                                         readSequence2.Add);
                sw3.Stop();

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

                Assert.True(sequence.SequenceEqual(readSequence2));

                // Assert.True(sw2.Elapsed < sw3.Elapsed);

                Console.WriteLine($"Stack-based walker: {sw3.Elapsed}, Level-based reader:
                ↪ {sw2.Elapsed}");

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

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

```
using System.IO;
using Xunit;
using Platform.Singletons;
using Platform.Memory;
using Platform.Data.Constants;
using Platform.Data.Doublets.ResizableDirectMemory;

namespace Platform.Data.Doublets.Tests
{
    public static class ResizableDirectMemoryLinksTests
    {
        private static readonly LinksCombinedConstants<ulong, ulong, int> _constants =
        ↪ Default<LinksCombinedConstants<ulong, ulong, int>>.Instance;

        [Fact]
        public static void BasicFileMappedMemoryTest()
        {
            var tempFilename = Path.GetTempFileName();
            using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(tempFilename))
            {
                memoryAdapter.TestBasicMemoryOperations();
            }
            File.Delete(tempFilename);
        }

        [Fact]
        public static void BasicHeapMemoryTest()
        {
            using (var memory = new
            ↪ HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
            using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
            ↪ UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
            {
                memoryAdapter.TestBasicMemoryOperations();
            }
        }

        private static void TestBasicMemoryOperations(this ILinks<ulong> memoryAdapter)
        {
```

```
            var link = memoryAdapter.Create();
            memoryAdapter.Delete(link);
        }

        [Fact]
        public static void NonexistentReferencesHeapMemoryTest()
        {
            using (var memory = new
            ↪  HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
            using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
            ↪  UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
            {
                memoryAdapter.TestNonexistentReferences();
            }
        }

        private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
        {
            var link = memoryAdapter.Create();
            memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
            var resultLink = _constants.Null;
            memoryAdapter.Each(foundLink =>
            {
                resultLink = foundLink[_constants.IndexPart];
                return _constants.Break;
            }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
            Assert.True(resultLink == link);
            Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
            memoryAdapter.Delete(link);
        }
    }
}
```

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

```
using Xunit;
using Platform.Scopes;
using Platform.Memory;
using Platform.Data.Doublets.ResizableDirectMemory;
using Platform.Data.Doublets.Decorators;

namespace Platform.Data.Doublets.Tests
{
    public static class ScopeTests
    {
        [Fact]
        public static void SingleDependencyTest()
        {
            using (var scope = new Scope())
            {
                scope.IncludeAssemblyOf<IMemory>();
                var instance = scope.Use<IDirectMemory>();
                Assert.IsType<HeapResizableDirectMemory>(instance);
            }
        }

        [Fact]
        public static void CascadeDependencyTest()
        {
            using (var scope = new Scope())
            {
                scope.Include<TemporaryFileMappedResizableDirectMemory>();
                scope.Include<UInt64ResizableDirectMemoryLinks>();
                var instance = scope.Use<ILinks<ulong>>();
                Assert.IsType<UInt64ResizableDirectMemoryLinks>(instance);
            }
        }

        [Fact]
        public static void FullAutoResolutionTest()
        {
            using (var scope = new Scope(autoInclude: true, autoExplore: true))
            {
                var instance = scope.Use<UInt64Links>();
                Assert.IsType<UInt64Links>(instance);
            }
        }
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using Xunit;
using Platform.Collections;
using Platform.Random;
using Platform.IO;
using Platform.Singletons;
using Platform.Data.Constants;
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Unicode;

namespace Platform.Data.Doublets.Tests
{
    public static class SequencesTests
    {
        private static readonly LinksCombinedConstants<bool, ulong, int> _constants = Default<LinksCombinedConstants<bool, ulong, int>>.Instance;

        static SequencesTests()
        {
            // Trigger static constructor to not mess with perfomance measurements
            _ = BitString.GetBitMaskFromIndex(1);
        }

        [Fact]
        public static void CreateAllVariantsTest()
        {
            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 sw1 = Stopwatch.StartNew();
                var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();

                var sw2 = Stopwatch.StartNew();
                var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();

                Assert.True(results1.Count > results2.Length);
                Assert.True(sw1.Elapsed > sw2.Elapsed);

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

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

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

        //    const long sequenceLength = 8;

        //    const ulong itself = LinksConstants.Itself;

        //    using (var memoryAdapter = new ResizableDirectMemoryLinks(tempFilename, DefaultLinksSizeStep))
        //    using (var links = new Links(memoryAdapter))
        //    {
        //        var sequence = new ulong[sequenceLength];
        //        for (var i = 0; i < sequenceLength; i++)
        //            sequence[i] = links.Create(itself, itself);

```

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

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

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

```
            {
                var links = scope.Links;
                var sequences = scope.Sequences;

                var e1 = links.Create();
                var e2 = links.Create();

                var sequence = new[]
                {
                    e1, e2, e1, e2 // mama / papa
                };

                var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);

                var balancedVariant = balancedVariantConverter.Convert(sequence);

                // 1: [1]
                // 2: [2]
                // 3: [1,2]
                // 4: [1,2,1,2]

                var doublet = links.GetSource(balancedVariant);

                var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);

                Assert.True(matchedSequences1.Count == 0);

                var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);

                Assert.True(matchedSequences2.Count == 0);

                var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);

                Assert.True(matchedSequences3.Count == 0);

                var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);

                Assert.Contains(doublet, matchedSequences4);
                Assert.Contains(balancedVariant, matchedSequences4);

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

        [Fact]
        public static void IndexTest()
        {
            using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
             ↪  true }, useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;
                var index = sequences.Options.Index;

                var e1 = links.Create();
                var e2 = links.Create();

                var sequence = new[]
                {
                    e1, e2, e1, e2 // mama / papa
                };

                Assert.False(index.MightContain(sequence));

                index.Add(sequence);

                Assert.True(index.MightContain(sequence));
            }
        }

        /// <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>
        private static readonly string _exampleText =
            @"([english
             ↪  version](https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))
```

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

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?

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

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?

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?

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?

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?

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?

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

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

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

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

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

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

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

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

...

[![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animation-500.gif ""анимация"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animation-500.gif)";

        private static readonly string _exampleLoremIpsumText =
            @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.";

        [Fact]
        public static void CompressionTest()
        {
            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;

                var e1 = links.Create();
                var e2 = links.Create();

                var sequence = new[]
                {
                    e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
                };

                var balancedVariantConverter = new BalancedVariantConverter<ulong>(links.Unsync);
                var totalSequenceSymbolFrequencyCounter = new TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
                var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync, totalSequenceSymbolFrequencyCounter);
                var compressingConverter = new CompressingConverter<ulong>(links.Unsync, balancedVariantConverter, doubletFrequenciesCache);

```

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

```
523                 //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
                    ↪  frequencyPropertyOperator, frequencyIncrementer);
524                 //var linkToItsFrequencyNumberConverter = new
                    ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                    ↪  unaryNumberToAddressConverter);
525
526                 var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
                    ↪  totalSequenceSymbolFrequencyCounter);
527
528                 var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
                    ↪  ncyNumberConverter<ulong>(linkFrequenciesCache3);
529
530                 var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
                    ↪  linkToItsFrequencyNumberConverter);
531                 var optimalVariantConverter = new
                    ↪  OptimalVariantConverter<ulong>(scope3.Links.Unsync,
                    ↪  sequenceToItsLocalElementLevelsConverter);
532
533                 var compressed1 = new ulong[arrays.Length];
534                 var compressed2 = new ulong[arrays.Length];
535                 var compressed3 = new ulong[arrays.Length];
536
537                 var START = 0;
538                 var END = arrays.Length;
539
540                 //for (int i = START; i < END; i++)
541                 //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
542
543                 var initialCount1 = scope2.Links.Unsync.Count();
544
545                 var sw1 = Stopwatch.StartNew();
546
547                 for (int i = START; i < END; i++)
548                 {
549                     linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
550                     compressed1[i] = compressor1.Convert(arrays[i]);
551                 }
552
553                 var elapsed1 = sw1.Elapsed;
554
555                 var balancedVariantConverter2 = new
                    ↪  BalancedVariantConverter<ulong>(scope2.Links.Unsync);
556
557                 var initialCount2 = scope2.Links.Unsync.Count();
558
559                 var sw2 = Stopwatch.StartNew();
560
561                 for (int i = START; i < END; i++)
562                 {
563                     compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
564                 }
565
566                 var elapsed2 = sw2.Elapsed;
567
568                 for (int i = START; i < END; i++)
569                 {
570                     linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
571                 }
572
573                 var initialCount3 = scope3.Links.Unsync.Count();
574
575                 var sw3 = Stopwatch.StartNew();
576
577                 for (int i = START; i < END; i++)
578                 {
579                     //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
580                     compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
581                 }
582
583                 var elapsed3 = sw3.Elapsed;
584
585                 Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
                    ↪  Optimal variant: {elapsed3}");
586
587                 // Assert.True(elapsed1 > elapsed2);
588
589                 // Checks
590                 for (int i = START; i < END; i++)
```

```
591                 {
592                     var sequence1 = compressed1[i];
593                     var sequence2 = compressed2[i];
594                     var sequence3 = compressed3[i];
595 
596                     var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                        ↪  scope1.Links.Unsync);
597 
598                     var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                        ↪  scope2.Links.Unsync);
599 
600                     var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
                        ↪  scope3.Links.Unsync);
601 
602                     var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
                        ↪  link.IsPartialPoint());
603                     var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
                        ↪  link.IsPartialPoint());
604                     var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
                        ↪  link.IsPartialPoint());
605 
606                     //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
                        ↪  arrays[i].Length > 3)
607                     //    Assert.False(structure1 == structure2);
608                     //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
                        ↪  arrays[i].Length > 3)
609                     //    Assert.False(structure3 == structure2);
610 
611                     Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
612                     Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
613                 }
614 
615                 Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
                    ↪  totalCharacters);
616                 Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
                    ↪  totalCharacters);
617                 Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
                    ↪  totalCharacters);
618 
619                 Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
                    ↪  totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) /
                    ↪  totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) /
                    ↪  totalCharacters}");
620 
621                 Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
                    ↪  scope2.Links.Unsync.Count() - initialCount2);
622                 Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
                    ↪  scope2.Links.Unsync.Count() - initialCount2);
623 
624                 var duplicateProvider1 = new
                    ↪  DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
625                 var duplicateProvider2 = new
                    ↪  DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
626                 var duplicateProvider3 = new
                    ↪  DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
627 
628                 var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
629                 var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
630                 var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
631 
632                 var duplicates1 = duplicateCounter1.Count();
633 
634                 ConsoleHelpers.Debug("------");
635 
636                 var duplicates2 = duplicateCounter2.Count();
637 
638                 ConsoleHelpers.Debug("------");
639 
640                 var duplicates3 = duplicateCounter3.Count();
641 
642                 Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
643 
644                 linkFrequenciesCache1.ValidateFrequencies();
645                 linkFrequenciesCache3.ValidateFrequencies();
646             }
647         }
648 
649         [Fact]
```

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

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

```
798             using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
                ↪  SequencesOptions<ulong> { UseCompression = true,
                ↪  EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
799             using (var scope2 = new TempLinksTestScope(useSequences: true))
800             {
801                 scope1.Links.UseUnicode();
802                 scope2.Links.UseUnicode();
803
804                 var compressor1 = scope1.Sequences;
805                 var compressor2 = scope2.Sequences;
806
807                 var compressed1 = new ulong[arrays.Length];
808                 var compressed2 = new ulong[arrays.Length];
809
810                 var sw1 = Stopwatch.StartNew();
811
812                 var START = 0;
813                 var END = arrays.Length;
814
815                 for (int i = START; i < END; i++)
816                 {
817                     compressed1[i] = compressor1.Create(arrays[i]);
818                 }
819
820                 var elapsed1 = sw1.Elapsed;
821
822                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
823
824                 var sw2 = Stopwatch.StartNew();
825
826                 for (int i = START; i < END; i++)
827                 {
828                     compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
829                 }
830
831                 var elapsed2 = sw2.Elapsed;
832
833                 Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
                    ↪  {elapsed2}");
834
835                 Assert.True(elapsed1 > elapsed2);
836
837                 // Checks
838                 for (int i = START; i < END; i++)
839                 {
840                     var sequence1 = compressed1[i];
841                     var sequence2 = compressed2[i];
842
843                     if (sequence1 != _constants.Null && sequence2 != _constants.Null)
844                     {
845                         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                            ↪  scope1.Links);
846
847                         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                            ↪  scope2.Links);
848
849                         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
850                     }
851                 }
852
853                 Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
854                 Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
855
856                 Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
                    ↪  totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
                    ↪  totalCharacters}");
857
858                 // Can be worse than balanced variant
859                 //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
860
861                 //compressor1.ValidateFrequencies();
862             }
863         }
864
865         [Fact]
866         public static void AllTreeBreakDownAtSequencesCreationBugTest()
867         {
868             // Made out of AllPossibleConnectionsTest test.
869
```

```
            //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);
                }

                for (var i = 0; i < sequenceLength; i++)
                {
```

```
                    links.Delete(sequence[i]);
                }
            }
        }

        [Fact(Skip = "Correct implementation is pending")]
        public static void CalculateAllUsagesTest()
        {
            const long sequenceLength = 3;

            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 linksTotalUsages1 = new ulong[links.Count() + 1];

                    sequences.CalculateAllUsages(linksTotalUsages1);

                    var linksTotalUsages2 = new ulong[links.Count() + 1];

                    sequences.CalculateAllUsages2(linksTotalUsages2);

                    var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
                    Assert.True(intersection1.Count == linksTotalUsages2.Length);
                }

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

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

```
using System.IO;
using Platform.Disposables;
using Platform.Data.Doublets.ResizableDirectMemory;
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Decorators;

namespace Platform.Data.Doublets.Tests
{
    public class TempLinksTestScope : DisposableBase
    {
        public readonly ILinks<ulong> MemoryAdapter;
        public readonly SynchronizedLinks<ulong> Links;
        public readonly Sequences.Sequences Sequences;
        public readonly string TempFilename;
        public readonly string TempTransactionLogFilename;
        private readonly bool _deleteFiles;

        public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool
        ↪  useLog = false)
            : this(new SequencesOptions<ulong>(), deleteFiles, useSequences, useLog)
        {
        }

        public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles =
        ↪  true, bool useSequences = false, bool useLog = false)
        {
            _deleteFiles = deleteFiles;
            TempFilename = Path.GetTempFileName();
            TempTransactionLogFilename = Path.GetTempFileName();

```

```
            var coreMemoryAdapter = new UInt64ResizableDirectMemoryLinks(TempFilename);

            MemoryAdapter = useLog ? (ILinks<ulong>)new
            ↪  UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
            ↪  coreMemoryAdapter;

            Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
            if (useSequences)
            {
                Sequences = new Sequences.Sequences(Links, sequencesOptions);
            }
        }

        protected override void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                Links.Unsync.DisposeIfPossible();
                if (_deleteFiles)
                {
                    DeleteFiles();
                }
            }
        }

        public void DeleteFiles()
        {
            File.Delete(TempFilename);
            File.Delete(TempTransactionLogFilename);
        }
    }
}
```

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

```
using System.Collections.Generic;
using Xunit;
using Platform.Ranges;
using Platform.Numbers;
using Platform.Random;
using Platform.Setters;

namespace Platform.Data.Doublets.Tests
{
    public static class TestExtensions
    {
        public static void TestCRUDOperations<T>(this ILinks<T> links)
        {
            var constants = links.Constants;

            var equalityComparer = EqualityComparer<T>.Default;

            // Create Link
            Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));

            var setter = new Setter<T>(constants.Null);
            links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);

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

            var linkAddress = links.Create();

            var link = new Link<T>(links.GetLink(linkAddress));

            Assert.True(link.Count == 3);
            Assert.True(equalityComparer.Equals(link.Index, linkAddress));
            Assert.True(equalityComparer.Equals(link.Source, constants.Null));
            Assert.True(equalityComparer.Equals(link.Target, constants.Null));

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

            // Get first link
            setter = new Setter<T>(constants.Null);
            links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);

            Assert.True(equalityComparer.Equals(setter.Result, linkAddress));

            // Update link to reference itself
            links.Update(linkAddress, linkAddress, linkAddress);

            link = new Link<T>(links.GetLink(linkAddress));
```

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

```
127
128             // Update link to reference null (prepare for delete)
129             var updated = links.Update(linkAddress3, constants.Null, constants.Null);
130
131             Assert.True(equalityComparer.Equals(updated, linkAddress3));
132
133             link3 = new Link<T>(links.GetLink(linkAddress3));
134
135             Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
136             Assert.True(equalityComparer.Equals(link3.Target, constants.Null));
137
138             // Delete link
139             links.Delete(linkAddress3);
140
141             Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Two));
142
143             var setter3 = new Setter<T>(constants.Null);
144             links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);
145
146             Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
147         }
148
149         public static void TestMultipleRandomCreationsAndDeletions<TLink>(this ILinks<TLink>
            ↪ links, int maximumOperationsPerCycle)
150         {
151             var comparer = Comparer<TLink>.Default;
152             for (var N = 1; N < maximumOperationsPerCycle; N++)
153             {
154                 var random = new System.Random(N);
155                 var created = 0;
156                 var deleted = 0;
157                 for (var i = 0; i < N; i++)
158                 {
159                     long linksCount = (Integer<TLink>)links.Count();
160                     var createPoint = random.NextBoolean();
161                     if (linksCount > 2 && createPoint)
162                     {
163                         var linksAddressRange = new Range<ulong>(1, (ulong)linksCount);
164                         TLink source = (Integer<TLink>)random.NextUInt64(linksAddressRange);
165                         TLink target = (Integer<TLink>)random.NextUInt64(linksAddressRange);
                            ↪ //-V3086
166                         var resultLink = links.CreateAndUpdate(source, target);
167                         if (comparer.Compare(resultLink, (Integer<TLink>)linksCount) > 0)
168                         {
169                             created++;
170                         }
171                     }
172                     else
173                     {
174                         links.Create();
175                         created++;
176                     }
177                 }
178                 Assert.True(created == (Integer<TLink>)links.Count());
179                 for (var i = 0; i < N; i++)
180                 {
181                     TLink link = (Integer<TLink>)(i + 1);
182                     if (links.Exists(link))
183                     {
184                         links.Delete(link);
185                         deleted++;
186                     }
187                 }
188                 Assert.True((Integer<TLink>)links.Count() == 0);
189             }
190         }
191     }
192 }
```

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.IO;
5  using System.Text;
6  using System.Threading;
7  using System.Threading.Tasks;
8  using Xunit;
9  using Platform.Disposables;
10 using Platform.IO;
```

```
11  using Platform.Ranges;
12  using Platform.Random;
13  using Platform.Timestamps;
14  using Platform.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 UInt64LinksTests
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              using (var scope = new TempLinksTestScope())
35              {
36                  scope.Links.TestMultipleRandomCreationsAndDeletions(100);
37              }
38          }
39
40          [Fact]
41          public static void CascadeUpdateTest()
42          {
43              var itself = _constants.Itself;
44
45              using (var scope = new TempLinksTestScope(useLog: true))
46              {
47                  var links = scope.Links;
48
49                  var l1 = links.Create();
50                  var l2 = links.Create();
51
52                  l2 = links.Update(l2, l2, l1, l2);
53
54                  links.CreateAndUpdate(l2, itself);
55                  links.CreateAndUpdate(l2, itself);
56
57                  l2 = links.Update(l2, l1);
58
59                  links.Delete(l2);
60
61                  Global.Trash = links.Count();
62
63                  links.Unsync.DisposeIfPossible(); // Close links to access log
64
65                  Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop
                    ↪  e.TempTransactionLogFilename);
66              }
67          }
68
69          [Fact]
70          public static void BasicTransactionLogTest()
71          {
72              using (var scope = new TempLinksTestScope(useLog: true))
73              {
74                  var links = scope.Links;
75                  var l1 = links.Create();
76                  var l2 = links.Create();
77
78                  Global.Trash = links.Update(l2, l2, l1, l2);
79
80                  links.Delete(l1);
81
82                  links.Unsync.DisposeIfPossible(); // Close links to access log
83
84                  Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop
                    ↪  e.TempTransactionLogFilename);
85              }
86          }
87
88          [Fact]
```

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

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

```
236
237                     links.Delete(l1);
238
239                     transaction.Commit();
240                 }
241
242                 Global.Trash = links.Count();
243             }
244
245             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                ↪ sactionLogFilename);
246         }
247
248         [Fact]
249         public static void TransactionDamage()
250         {
251             var itself = _constants.Itself;
252
253             var tempDatabaseFilename = Path.GetTempFileName();
254             var tempTransactionLogFilename = Path.GetTempFileName();
255
256             // Commit
257             using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                ↪ tempTransactionLogFilename))
258             using (var links = new UInt64Links(memoryAdapter))
259             {
260                 using (var transaction = memoryAdapter.BeginTransaction())
261                 {
262                     var l1 = links.CreateAndUpdate(itself, itself);
263                     var l2 = links.CreateAndUpdate(itself, itself);
264
265                     Global.Trash = links.Update(l2, l2, l1, l2);
266
267                     links.Delete(l1);
268
269                     transaction.Commit();
270                 }
271
272                 Global.Trash = links.Count();
273             }
274
275             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                ↪ sactionLogFilename);
276
277             // Damage database
278
279             FileHelpers.WriteFirst(tempTransactionLogFilename, new
                ↪ UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
280
281             // Try load damaged database
282             try
283             {
284                 // TODO: Fix
285                 using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                    ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                    ↪ tempTransactionLogFilename))
286                 using (var links = new UInt64Links(memoryAdapter))
287                 {
288                     Global.Trash = links.Count();
289                 }
290             }
291             catch (NotSupportedException ex)
292             {
293                 Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
                    ↪ yet.");
294             }
295
296             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                ↪ sactionLogFilename);
297
298             File.Delete(tempDatabaseFilename);
299             File.Delete(tempTransactionLogFilename);
300         }
301
302         [Fact]
303         public static void Bug1Test()
304         {
```

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

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

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

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

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

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

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

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

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

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

        #endregion

        #region Performance

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

            return;

            try
            {
                //ThreadPool.SetMaxThreads(2, 2);

                // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
    ↪  результат
                // Также это дополнительно помогает в отладке
                // Увеличивает вероятность попадания информации в кэши
                for (var i = 0; i < 10; i++)
                {
                    //0 - 10 ГБ
                    //Каждые 100 МБ срез цифр

```

```
                    //links.TestGetSourceFunction();
                    //links.TestGetSourceFunctionInParallel();
                    //links.TestGetTargetFunction();
                    //links.TestGetTargetFunctionInParallel();
                    links.Create64BillionLinks();

                    links.TestRandomSearchFixed();
                    //links.Create64BillionLinksInParallel();
                    links.TestEachFunction();
                    //links.TestForeach();
                    //links.TestParallelForeach();
                }

                links.TestDeletionOfAllLinks();

            }
            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);
```

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

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

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

```
                    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: 0(0), was dependent on creation tests")]
        public static void TestEach()
        {
            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;

                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;
```

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

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

```
1  using Xunit;
2  using Platform.Random;
3  using Platform.Data.Doublets.Numbers.Unary;
4
5  namespace Platform.Data.Doublets.Tests
6  {
7      public static class UnaryNumberConvertersTests
8      {
9          [Fact]
10         public static void ConvertersTest()
11         {
12             using (var scope = new TempLinksTestScope())
13             {
14                 const int N = 10;
15                 var links = scope.Links;
16                 var meaningRoot = links.CreatePoint();
17                 var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
18                 var powerOf2ToUnaryNumberConverter = new
                    ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, one);
19                 var toUnaryNumberConverter = new AddressToUnaryNumberConverter<ulong>(links,
                    ↪ powerOf2ToUnaryNumberConverter);
20                 var random = new System.Random(0);
21                 ulong[] numbers = new ulong[N];
22                 ulong[] unaryNumbers = new ulong[N];
23                 for (int i = 0; i < N; i++)
24                 {
25                     numbers[i] = random.NextUInt64();
26                     unaryNumbers[i] = toUnaryNumberConverter.Convert(numbers[i]);
27                 }
28                 var fromUnaryNumberConverterUsingOrOperation = new
                    ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links,
                    ↪ powerOf2ToUnaryNumberConverter);
29                 var fromUnaryNumberConverterUsingAddOperation = new
                    ↪ UnaryNumberToAddressAddOperationConverter<ulong>(links, one);
30                 for (int i = 0; i < N; i++)
31                 {
32                     Assert.Equal(numbers[i],
                        ↪ fromUnaryNumberConverterUsingOrOperation.Convert(unaryNumbers[i]));
33                     Assert.Equal(numbers[i],
                        ↪ fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));
34                 }
35             }
36         }
37     }
38 }
```

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

```
using Xunit;
using Platform.Interfaces;
using Platform.Memory;
using Platform.Reflection;
using Platform.Scopes;
using Platform.Data.Doublets.Incrementers;
using Platform.Data.Doublets.Numbers.Raw;
using Platform.Data.Doublets.Numbers.Unary;
using Platform.Data.Doublets.PropertyOperators;
using Platform.Data.Doublets.ResizableDirectMemory;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Sequences.Indexes;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Unicode;

namespace Platform.Data.Doublets.Tests
{
    public static class UnicodeConvertersTests
    {
        [Fact]
        public static void CharAndUnaryNumberUnicodeSymbolConvertersTest()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                var meaningRoot = links.CreatePoint();
                var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
                var powerOf2ToUnaryNumberConverter = new
                ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, one);
                var addressToUnaryNumberConverter = new
                ↪  AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
                var unaryNumberToAddressConverter = new
                ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links,
                ↪  powerOf2ToUnaryNumberConverter);
                TestCharAndUnicodeSymbolConverters(links, meaningRoot,
                ↪  addressToUnaryNumberConverter, unaryNumberToAddressConverter);
            }
        }

        [Fact]
        public static void CharAndRawNumberUnicodeSymbolConvertersTest()
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪  ResizableDirectMemoryLinks<ulong>>>())
            {
                var links = scope.Use<ILinks<ulong>>();
                var meaningRoot = links.CreatePoint();
                var addressToRawNumberConverter = new AddressToRawNumberConverter<ulong>();
                var rawNumberToAddressConverter = new RawNumberToAddressConverter<ulong>();
                TestCharAndUnicodeSymbolConverters(links, meaningRoot,
                ↪  addressToRawNumberConverter, rawNumberToAddressConverter);
            }
        }

        private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
        ↪  meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
        ↪  numberToAddressConverter)
        {
            var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
            var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links,
            ↪  addressToNumberConverter, unicodeSymbolMarker);
            var originalCharacter = 'H';
            var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);
            var unicodeSymbolCriterionMatcher = new UnicodeSymbolCriterionMatcher<ulong>(links,
            ↪  unicodeSymbolMarker);
            var unicodeSymbolToCharConverter = new UnicodeSymbolToCharConverter<ulong>(links,
            ↪  numberToAddressConverter, unicodeSymbolCriterionMatcher);
            var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
            Assert.Equal(originalCharacter, resultingCharacter);
        }

        [Fact]
        public static void StringAndUnicodeSequenceConvertersTest()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;

```

```
67                  var itself = links.Constants.Itself;
68
69                  var meaningRoot = links.CreatePoint();
70                  var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
71                  var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
72                  var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
73                  var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
74                  var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);
75
76                  var powerOf2ToUnaryNumberConverter = new
                    ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
77                  var addressToUnaryNumberConverter = new
                    ↪  AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
78                  var charToUnicodeSymbolConverter = new
                    ↪  CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
                    ↪  unicodeSymbolMarker);
79
80                  var unaryNumberToAddressConverter = new
                    ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links,
                    ↪  powerOf2ToUnaryNumberConverter);
81                  var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
82                  var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                    ↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
83                  var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
                    ↪  frequencyPropertyMarker, frequencyMarker);
84                  var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
                    ↪  frequencyPropertyOperator, frequencyIncrementer);
85                  var linkToItsFrequencyNumberConverter = new
                    ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                    ↪  unaryNumberToAddressConverter);
86                  var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                    ↪  linkToItsFrequencyNumberConverter);
87                  var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                    ↪  sequenceToItsLocalElementLevelsConverter);
88
89                  var stringToUnicodeSymbolConverter = new
                    ↪  StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
                    ↪  index, optimalVariantConverter, unicodeSequenceMarker);
90
91                  var originalString = "Hello";
92
93                  var unicodeSequenceLink = stringToUnicodeSymbolConverter.Convert(originalString);
94
95                  var unicodeSymbolCriterionMatcher = new
                    ↪  UnicodeSymbolCriterionMatcher<ulong>(links, unicodeSymbolMarker);
96                  var unicodeSymbolToCharConverter = new
                    ↪  UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
                    ↪  unicodeSymbolCriterionMatcher);
97
98                  var unicodeSequenceCriterionMatcher = new
                    ↪  UnicodeSequenceCriterionMatcher<ulong>(links, unicodeSequenceMarker);
99
100                 var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
                    ↪  unicodeSymbolCriterionMatcher.IsMatched);
101
102                 var unicodeSequenceToStringConverter = new
                    ↪  UnicodeSequenceToStringConverter<ulong>(links,
                    ↪  unicodeSequenceCriterionMatcher, sequenceWalker,
                    ↪  unicodeSymbolToCharConverter);
103
104                 var resultingString =
                    ↪  unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
105
106                 Assert.Equal(originalString, resultingString);
107             }
108         }
109     }
110 }
```

Index