./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)
        {
            Links.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)
        {
            this.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; }
        protected LinksDecoratorBase(ILinks<TLink> links) : base(links) => Constants =
        ↪  links.Constants;
        public virtual TLink Count(IList<TLink> restriction) => Links.Count(restriction);
        public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
        ↪  => Links.Each(handler, restrictions);
        public virtual TLink Create() => Links.Create();
        public virtual TLink Update(IList<TLink> restrictions) => Links.Update(restrictions);
        public virtual void Delete(TLink link) => Links.Delete(link);
    }
}
```

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

```
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))
            {
                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>
    {
        public UInt64Links(ILinks<ulong> links) : base(links) { }

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

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

        public override ulong Update(IList<ulong> restrictions)
        {
            var constants = Constants;
            var nullConstant = constants.Null;
            if (restrictions.IsNullOrEmpty())
            {
                return nullConstant;
            }
            // TODO: Looks like this is a common type of exceptions linked with restrictions support
            if (restrictions.Count != 3)
            {
                throw new NotSupportedException();
            }
```

```
            var indexPartConstant = constants.IndexPart;
            var updatedLink = restrictions[indexPartConstant];
            this.EnsureLinkExists(updatedLink,
            ↪  $"{nameof(restrictions)}[{nameof(indexPartConstant)}]");
            var sourcePartConstant = constants.SourcePart;
            var newSource = restrictions[sourcePartConstant];
            this.EnsureLinkIsItselfOrExists(newSource,
            ↪  $"{nameof(restrictions)}[{nameof(sourcePartConstant)}]");
            var targetPartConstant = constants.TargetPart;
            var newTarget = restrictions[targetPartConstant];
            this.EnsureLinkIsItselfOrExists(newTarget,
            ↪  $"{nameof(restrictions)}[{nameof(targetPartConstant)}]");
            var existedLink = nullConstant;
            var itselfConstant = constants.Itself;
            if (newSource != itselfConstant && newTarget != itselfConstant)
            {
                existedLink = this.SearchOrDefault(newSource, newTarget);
            }
            if (existedLink == nullConstant)
            {
                var before = Links.GetLink(updatedLink);
                if (before[sourcePartConstant] != newSource || before[targetPartConstant] !=
                ↪  newTarget)
                {
                    Links.Update(updatedLink, newSource == itselfConstant ? updatedLink :
                    ↪  newSource,
                                              newTarget == itselfConstant ? updatedLink :
                                              ↪  newTarget);
                }
                return updatedLink;
            }
            else
            {
                return this.MergeAndDelete(updatedLink, existedLink);
            }
        }

        public override void Delete(ulong linkIndex)
        {
            Links.EnsureLinkExists(linkIndex);
            Links.EnforceResetValues(linkIndex);
            this.DeleteAllUsages(linkIndex);
            Links.Delete(linkIndex);
        }
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Linq;
using Platform.Collections;
using Platform.Collections.Arrays;
using Platform.Collections.Lists;
using Platform.Data.Universal;

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

namespace Platform.Data.Doublets.Decorators
{
    /// <remarks>
    /// What does empty pattern (for condition or substitution) mean? Nothing or Everything?
    /// 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).
    ///
    /// 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.)
    /// </remarks>
    internal class UniLinks<TLink> : LinksDecoratorBase<TLink>, IUniLinks<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

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

        private struct Transition
        {
            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;
72              ////                    newValue[Constants.SourcePart] =
                ↪  Equals(substitution[Constants.SourcePart], Constants.Itself) ?
                ↪  matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
73              ////                    newValue[Constants.TargetPart] =
                ↪  Equals(substitution[Constants.TargetPart], Constants.Itself) ?
                ↪  matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
74              ////                    var matchDecision = matchedHandler(matchedLink, newValue);
75              ////                    if (Equals(matchDecision, Constants.Break))
76              ////                        return false;
77              ////                    if (!Equals(matchDecision, Constants.Skip))
78              ////                        transitions.Add(new Transition(matchedLink, newValue));
79              ////                    return true;
80              ////                };
81              ////                if (!Memory.Each(handler, restriction))
82              ////                    return Constants.Break;
83              ////            }
84              ////        }
85              ////        else
86              ////        {
87              ////            Func<T, bool> handler = link =>
88              ////            {
89              ////                var matchedLink = Memory.GetLinkValue(link);
90              ////                var matchDecision = matchedHandler(matchedLink, matchedLink);
91              ////                return !Equals(matchDecision, Constants.Break);
92              ////            };
```

```
93              ////            if (!Memory.Each(handler, restriction))
94              ////                return Constants.Break;
95              ////        }
96              ////    }
97              ////    else
98              ////    {
99              ////        if (substitution != null)
100             ////        {
101             ////            transitions = new List<IList<T>>();
102             ////            Func<T, bool> handler = link =>
103             ////            {
104             ////                var matchedLink = Memory.GetLinkValue(link);
105             ////                transitions.Add(matchedLink);
106             ////                return true;
107             ////            };
108             ////            if (!Memory.Each(handler, restriction))
109             ////                return Constants.Break;
110             ////        }
111             ////        else
112             ////        {
113             ////            return Constants.Continue;
114             ////        }
115             ////    }
116             ////}
117             ////if (substitution != null)
118             ////{
119             ////    // Есть причина делать замену (запись)
120             ////    if (substitutedHandler != null)
121             ////    {
122             ////    }
123             ////    else
124             ////    {
125             ////    }
126             ////}
127             ////return Constants.Continue;
128
129             //if (restriction.IsNullOrEmpty()) // Create
130             //{
131             //    substitution[Constants.IndexPart] = Memory.AllocateLink();
132             //    Memory.SetLinkValue(substitution);
133             //}
134             //else if (substitution.IsNullOrEmpty()) // Delete
135             //{
136             //    Memory.FreeLink(restriction[Constants.IndexPart]);
137             //}
138             //else if (restriction.EqualTo(substitution)) // Read or ("repeat" the state) // Each
139             //{
140             //    // No need to collect links to list
141             //    // Skip == Continue
142             //    // No need to check substituedHandler
143             //    if (!Memory.Each(link => !Equals(matchedHandler(Memory.GetLinkValue(link)),
                ↪  Constants.Break), restriction))
144             //        return Constants.Break;
145             //}
146             //else // Update
147             //{
148             //    //List<IList<T>> matchedLinks = null;
149             //    if (matchedHandler != null)
150             //    {
151             //        matchedLinks = new List<IList<T>>();
152             //        Func<T, bool> handler = link =>
153             //        {
154             //            var matchedLink = Memory.GetLinkValue(link);
155             //            var matchDecision = matchedHandler(matchedLink);
156             //            if (Equals(matchDecision, Constants.Break))
157             //                return false;
158             //            if (!Equals(matchDecision, Constants.Skip))
159             //                matchedLinks.Add(matchedLink);
160             //            return true;
161             //        };
162             //        if (!Memory.Each(handler, restriction))
163             //            return Constants.Break;
164             //    }
165             //    if (!matchedLinks.IsNullOrEmpty())
166             //    {
167             //        var totalMatchedLinks = matchedLinks.Count;
168             //        for (var i = 0; i < totalMatchedLinks; i++)
169             //        {
```

```
170             //            var matchedLink = matchedLinks[i];
171             //            if (substitutedHandler != null)
172             //            {
173             //                var newValue = new List<T>(); // TODO: Prepare value to update here
174             //                // TODO: Decide is it actually needed to use Before and After
                ↪  substitution handling.
175             //                var substitutedDecision = substitutedHandler(matchedLink,
                ↪  newValue);
176             //                if (Equals(substitutedDecision, Constants.Break))
177             //                    return Constants.Break;
178             //                if (Equals(substitutedDecision, Constants.Continue))
179             //                {
180             //                    // Actual update here
181             //                    Memory.SetLinkValue(newValue);
182             //                }
183             //                if (Equals(substitutedDecision, Constants.Skip))
184             //                {
185             //                    // Cancel the update. TODO: decide use separate Cancel
                ↪  constant or Skip is enough?
186             //                }
187             //            }
188             //        }
189             //    }
190             //}
191             return Constants.Continue;
192         }
193
194         public TLink Trigger(IList<TLink> patternOrCondition, Func<IList<TLink>, TLink>
            ↪  matchHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
            ↪  substitutionHandler)
195         {
196             if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
197             {
198                 return Constants.Continue;
199             }
200             else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO:
                ↪  Check if it is a correct condition
201             {
202                 // Or it only applies to trigger without matchHandler.
203                 throw new NotImplementedException();
204             }
205             else if (!substitution.IsNullOrEmpty()) // Creation
206             {
207                 var before = ArrayPool<TLink>.Empty;
208                 // Что должно означать False здесь? Остановиться (перестать идти) или пропустить
                    ↪  (пройти мимо) или пустить (взять)?
209                 if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                    ↪  Constants.Break))
210                 {
211                     return Constants.Break;
212                 }
213                 var after = (IList<TLink>)substitution.ToArray();
214                 if (_equalityComparer.Equals(after[0], default))
215                 {
216                     var newLink = Links.Create();
217                     after[0] = newLink;
218                 }
219                 if (substitution.Count == 1)
220                 {
221                     after = Links.GetLink(substitution[0]);
222                 }
223                 else if (substitution.Count == 3)
224                 {
225                     Links.Update(after);
226                 }
227                 else
228                 {
229                     throw new NotSupportedException();
230                 }
231                 if (matchHandler != null)
232                 {
233                     return substitutionHandler(before, after);
234                 }
235                 return Constants.Continue;
236             }
237             else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
238             {
239                 if (patternOrCondition.Count == 1)
```

```
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
290                     {
291                         throw new NotSupportedException();
292                     }
293                     if (matchHandler != null)
294                     {
295                         return substitutionHandler(before, after);
296                     }
297                     return Constants.Continue;
298                 }
299                 else
300                 {
301                     throw new NotSupportedException();
302                 }
303             }
304         }
305 
306         /// <remarks>
307         /// IList[IList[IList[T]]]
308         /// |      |      |      |||
309         /// |      |      ------ ||
310         /// |      |       link  ||
311         /// |      -------------- |
312         /// |         change     |
313         ///  --------------------
314         ///         changes
315         /// </remarks>
```

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

./Platform.Data.Doublets/DoubletComparer.cs

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

./Platform.Data.Doublets/Doublet.cs

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

./Platform.Data.Doublets/Hybrid.cs

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

./Platform.Data.Doublets/ILinks.cs

```
using Platform.Data.Constants;

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

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

./Platform.Data.Doublets/ILinksExtensions.cs

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

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

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

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

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

        /// <remarks>
        /// TODO: Возможно есть очень простой способ это сделать.
```

```
58          /// (Например просто удалить файл, или изменить его размер таким образом,
59          /// чтобы удалился весь контент)
60          /// Например через _header->AllocatedLinks в ResizableDirectMemoryLinks
61          /// </remarks>
62          public static void DeleteAll<TLink>(this ILinks<TLink> links)
63          {
64              var equalityComparer = EqualityComparer<TLink>.Default;
65              var comparer = Comparer<TLink>.Default;
66              for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
                ↪ Arithmetic.Decrement(i))
67              {
68                  links.Delete(i);
69                  if (!equalityComparer.Equals(links.Count(), Arithmetic.Decrement(i)))
70                  {
71                      i = links.Count();
72                  }
73              }
74          }
75
76          public static TLink First<TLink>(this ILinks<TLink> links)
77          {
78              TLink firstLink = default;
79              var equalityComparer = EqualityComparer<TLink>.Default;
80              if (equalityComparer.Equals(links.Count(), default))
81              {
82                  throw new Exception("В хранилище нет связей.");
83              }
84              links.Each(links.Constants.Any, links.Constants.Any, link =>
85              {
86                  firstLink = link[links.Constants.IndexPart];
87                  return links.Constants.Break;
88              });
89              if (equalityComparer.Equals(firstLink, default))
90              {
91                  throw new Exception("В процессе поиска по хранилищу не было найдено связей.");
92              }
93              return firstLink;
94          }
95
96          public static bool IsInnerReference<TLink>(this ILinks<TLink> links, TLink reference)
97          {
98              var constants = links.Constants;
99              var comparer = Comparer<TLink>.Default;
100             return comparer.Compare(constants.MinPossibleIndex, reference) >= 0 &&
                ↪ comparer.Compare(reference, constants.MaxPossibleIndex) <= 0;
101         }
102
103         #region Paths
104
105         /// <remarks>
106         /// TODO: Как так? Как то что ниже может быть корректно?
107         /// Скорее всего практически не применимо
108         /// Предполагалось, что можно было конвертировать формируемый в проходе через
                ↪ SequenceWalker
109         /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
110         /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
111         /// </remarks>
112         public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[]
                ↪ path)
113         {
114             var current = path[0];
115             //EnsureLinkExists(current, "path");
116             if (!links.Exists(current))
117             {
118                 return false;
119             }
120             var equalityComparer = EqualityComparer<TLink>.Default;
121             var constants = links.Constants;
122             for (var i = 1; i < path.Length; i++)
123             {
124                 var next = path[i];
125                 var values = links.GetLink(current);
126                 var source = values[constants.SourcePart];
127                 var target = values[constants.TargetPart];
128                 if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
                    ↪ next))
129                 {
130                     //throw new Exception(string.Format("Невозможно выбрать путь, так как и
                        ↪ Source и Target совпадают с элементом пути {0}.", next));
```

```
                    return false;
                }
                if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
                ↪  target))
                {
                    //throw new Exception(string.Format("Невозможно продолжить путь через
                    ↪  элемент пути {0}", next));
                    return false;
                }
                current = next;
            }
            return true;
        }

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

        public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this ILinks<TLink>
        ↪  links, TLink root, ulong size, ulong index)
        {
            var constants = links.Constants;
            var source = constants.SourcePart;
            var target = constants.TargetPart;
            if (!Numbers.Math.IsPowerOfTwo(size))
            {
                throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes other
                ↪  than powers of two are not supported.");
            }
            var path = new BitArray(BitConverter.GetBytes(index));
            var length = Bit.GetLowestPosition(size);
            links.EnsureLinkExists(root, "root");
            var currentLink = root;
            for (var i = length - 1; i >= 0; i--)
            {
                currentLink = links.GetLink(currentLink)[path[i] ? target : source];
            }
            return currentLink;
        }

        #endregion

        /// <summary>
        /// Возвращает индекс указанной связи.
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="link">Связь представленная списком, состоящим из её адреса и
        ↪  содержимого.</param>
        /// <returns>Индекс начальной связи для указанной связи.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
        ↪  link[links.Constants.IndexPart];

        /// <summary>
        /// Возвращает индекс начальной (Source) связи для указанной связи.
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="link">Индекс связи.</param>
        /// <returns>Индекс начальной связи для указанной связи.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink link) =>
        ↪  links.GetLink(link)[links.Constants.SourcePart];

        /// <summary>
        /// Возвращает индекс начальной (Source) связи для указанной связи.
        /// </summary>
```

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

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

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

```
            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)]
```

```
456         public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
457         {
458             var link = links.Create();
459             return links.Update(link, link, link);
460         }
461 
462         /// <param name="links">Хранилище связей.</param>
463         [MethodImpl(MethodImplOptions.AggressiveInlining)]
464         public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
            ↪ target) => links.Update(links.Create(), source, target);
465 
466         /// <summary>
467         /// Обновляет связь с указанными началом (Source) и концом (Target)
468         /// на связь с указанными началом (NewSource) и концом (NewTarget).
469         /// </summary>
470         /// <param name="links">Хранилище связей.</param>
471         /// <param name="link">Индекс обновляемой связи.</param>
472         /// <param name="newSource">Индекс связи, которая является началом связи, на которую
            ↪ выполняется обновление.</param>
473         /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
            ↪ выполняется обновление.</param>
474         /// <returns>Индекс обновлённой связи.</returns>
475         [MethodImpl(MethodImplOptions.AggressiveInlining)]
476         public static TLink Update<TLink>(this ILinks<TLink> links, TLink link, TLink newSource,
            ↪ TLink newTarget) => links.Update(new Link<TLink>(link, newSource, newTarget));
477 
478         /// <summary>
479         /// Обновляет связь с указанными началом (Source) и концом (Target)
480         /// на связь с указанными началом (NewSource) и концом (NewTarget).
481         /// </summary>
482         /// <param name="links">Хранилище связей.</param>
483         /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
            ↪ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
            ↪ Itself - требование установить ссылку на себя, 1..∞ конкретный адрес другой
            ↪ связи.</param>
484         /// <returns>Индекс обновлённой связи.</returns>
485         [MethodImpl(MethodImplOptions.AggressiveInlining)]
486         public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
487         {
488             if (restrictions.Length == 2)
489             {
490                 return links.MergeAndDelete(restrictions[0], restrictions[1]);
491             }
492             if (restrictions.Length == 4)
493             {
494                 return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
                    ↪ restrictions[2], restrictions[3]);
495             }
496             else
497             {
498                 return links.Update(restrictions);
499             }
500         }
501 
502         [MethodImpl(MethodImplOptions.AggressiveInlining)]
503         public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
            ↪ links, TLink constant, IList<TLink> restrictions)
504         {
505             var equalityComparer = EqualityComparer<TLink>.Default;
506             var constants = links.Constants;
507             var index = restrictions[constants.IndexPart];
508             var source = restrictions[constants.SourcePart];
509             var target = restrictions[constants.TargetPart];
510             source = equalityComparer.Equals(source, constant) ? index : source;
511             target = equalityComparer.Equals(target, constant) ? index : target;
512             return new Link<TLink>(index, source, target);
513         }
514 
515         /// <summary>
516         /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
            ↪ с указанными Source (началом) и Target (концом).
517         /// </summary>
518         /// <param name="links">Хранилище связей.</param>
519         /// <param name="source">Индекс связи, которая является началом на создаваемой
            ↪ связи.</param>
520         /// <param name="target">Индекс связи, которая является концом для создаваемой
            ↪ связи.</param>
```

```
521         /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
522         [MethodImpl(MethodImplOptions.AggressiveInlining)]
523         public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
            ↪ target)
524         {
525             var link = links.SearchOrDefault(source, target);
526             if (EqualityComparer<TLink>.Default.Equals(link, default))
527             {
528                 link = links.CreateAndUpdate(source, target);
529             }
530             return link;
531         }
532 
533         /// <summary>
534         /// Обновляет связь с указанными началом (Source) и концом (Target)
535         /// на связь с указанными началом (NewSource) и концом (NewTarget).
536         /// </summary>
537         /// <param name="links">Хранилище связей.</param>
538         /// <param name="source">Индекс связи, которая является началом обновляемой
            ↪ связи.</param>
539         /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
540         /// <param name="newSource">Индекс связи, которая является началом связи, на которую
            ↪ выполняется обновление.</param>
541         /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
            ↪ выполняется обновление.</param>
542         /// <returns>Индекс обновлённой связи.</returns>
543         [MethodImpl(MethodImplOptions.AggressiveInlining)]
544         public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
            ↪ TLink target, TLink newSource, TLink newTarget)
545         {
546             var equalityComparer = EqualityComparer<TLink>.Default;
547             var link = links.SearchOrDefault(source, target);
548             if (equalityComparer.Equals(link, default))
549             {
550                 return links.CreateAndUpdate(newSource, newTarget);
551             }
552             if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
                ↪ target))
553             {
554                 return link;
555             }
556             return links.Update(link, newSource, newTarget);
557         }
558 
559         /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
560         /// <param name="links">Хранилище связей.</param>
561         /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
562         /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
563         [MethodImpl(MethodImplOptions.AggressiveInlining)]
564         public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
            ↪ target)
565         {
566             var link = links.SearchOrDefault(source, target);
567             if (!EqualityComparer<TLink>.Default.Equals(link, default))
568             {
569                 links.Delete(link);
570                 return link;
571             }
572             return default;
573         }
574 
575         /// <summary>Удаляет несколько связей.</summary>
576         /// <param name="links">Хранилище связей.</param>
577         /// <param name="deletedLinks">Список адресов связей к удалению.</param>
578         [MethodImpl(MethodImplOptions.AggressiveInlining)]
579         public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
580         {
581             for (int i = 0; i < deletedLinks.Count; i++)
582             {
583                 links.Delete(deletedLinks[i]);
584             }
585         }
586 
587         /// <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>
588         public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
589         {
```

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

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

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

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

```
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();

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

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

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

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

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

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

```
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.Always.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.Always.ArgumentNotNull(array, nameof(array));
128             Ensure.Always.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;
134             array[arrayIndex++] = Source;
135             array[arrayIndex] = Target;
136         }
137
138         public bool Remove(TLink item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
139
140         public int IndexOf(TLink item)
141         {
142             if (_equalityComparer.Equals(Index, item))
143             {
144                 return _constants.IndexPart;
145             }
146             if (_equalityComparer.Equals(Source, item))
147             {
148                 return _constants.SourcePart;
149             }
150             if (_equalityComparer.Equals(Target, item))
```

```
            {
                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/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>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            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         {
181         }
182
183         public ResizableDirectMemoryLinks(IResizableDirectMemory memory)
```

```
184             : this(memory, DefaultLinksSizeStep)
185         {
186         }
187 
188         public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
            ↪ memoryReservationStep)
189         {
190             Constants = Default<LinksCombinedConstants<TLink, TLink, int>>.Instance;
191             _memory = memory;
192             _memoryReservationStep = memoryReservationStep;
193             if (memory.ReservedCapacity < memoryReservationStep)
194             {
195                 memory.ReservedCapacity = memoryReservationStep;
196             }
197             SetPointers(_memory);
198             // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
199             _memory.UsedCapacity = ((long)(Integer<TLink>)LinksHeader.GetAllocatedLinks(_header)
                ↪ * LinkSizeInBytes) + LinkHeaderSizeInBytes;
200             // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
201             LinksHeader.SetReservedLinks(_header, (Integer<TLink>)((_memory.ReservedCapacity -
                ↪ LinkHeaderSizeInBytes) / LinkSizeInBytes));
202         }
203 
204         [MethodImpl(MethodImplOptions.AggressiveInlining)]
205         public TLink Count(IList<TLink> restrictions)
206         {
207             // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
208             if (restrictions.Count == 0)
209             {
210                 return Total;
211             }
212             if (restrictions.Count == 1)
213             {
214                 var index = restrictions[Constants.IndexPart];
215                 if (_equalityComparer.Equals(index, Constants.Any))
216                 {
217                     return Total;
218                 }
219                 return Exists(index) ? Integer<TLink>.One : Integer<TLink>.Zero;
220             }
221             if (restrictions.Count == 2)
222             {
223                 var index = restrictions[Constants.IndexPart];
224                 var value = restrictions[1];
225                 if (_equalityComparer.Equals(index, Constants.Any))
226                 {
227                     if (_equalityComparer.Equals(value, Constants.Any))
228                     {
229                         return Total; // Any - как отсутствие ограничения
230                     }
231                     return Add(_sourcesTreeMethods.CountUsages(value),
                        ↪ _targetsTreeMethods.CountUsages(value));
232                 }
233                 else
234                 {
235                     if (!Exists(index))
236                     {
237                         return Integer<TLink>.Zero;
238                     }
239                     if (_equalityComparer.Equals(value, Constants.Any))
240                     {
241                         return Integer<TLink>.One;
242                     }
243                     var storedLinkValue = GetLinkUnsafe(index);
244                     if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
245                         _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
246                     {
247                         return Integer<TLink>.One;
248                     }
249                     return Integer<TLink>.Zero;
250                 }
251             }
252             if (restrictions.Count == 3)
253             {
254                 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                     }
401                     else //if(source != Any && target != Any)
402                     {
403                         var link = _sourcesTreeMethods.Search(source, target);
```

```
404                         return _equalityComparer.Equals(link, Constants.Null) ?
                            ↪  Constants.Continue : handler(GetLinkStruct(link));
405                     }
406                 }
407                 else
408                 {
409                     if (!Exists(index))
410                     {
411                         return Constants.Continue;
412                     }
413                     if (_equalityComparer.Equals(source, Constants.Any) &&
                        ↪  _equalityComparer.Equals(target, Constants.Any))
414                     {
415                         return handler(GetLinkStruct(index));
416                     }
417                     var storedLinkValue = GetLinkUnsafe(index);
418                     if (!_equalityComparer.Equals(source, Constants.Any) &&
                        ↪  !_equalityComparer.Equals(target, Constants.Any))
419                     {
420                         if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source) &&
421                             _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
422                         {
423                             return handler(GetLinkStruct(index));
424                         }
425                         return Constants.Continue;
426                     }
427                     var value = default(TLink);
428                     if (_equalityComparer.Equals(source, Constants.Any))
429                     {
430                         value = target;
431                     }
432                     if (_equalityComparer.Equals(target, Constants.Any))
433                     {
434                         value = source;
435                     }
436                     if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
437                         _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
438                     {
439                         return handler(GetLinkStruct(index));
440                     }
441                     return Constants.Continue;
442                 }
443             }
444             throw new NotSupportedException("Другие размеры и способы ограничений не
                ↪  поддерживаются.");
445         }
446 
447         /// <remarks>
448         /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
                ↪  в другом месте (но не в менеджере памяти, а в логике Links)
449         /// </remarks>
450         [MethodImpl(MethodImplOptions.AggressiveInlining)]
451         public TLink Update(IList<TLink> values)
452         {
453             var linkIndex = values[Constants.IndexPart];
454             var link = GetLinkUnsafe(linkIndex);
455             // Будет корректно работать только в том случае, если пространство выделенной связи
                ↪  предварительно заполнено нулями
456             if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
457             {
458                 _sourcesTreeMethods.Detach(LinksHeader.GetFirstAsSourcePointer(_header),
                    ↪  linkIndex);
459             }
460             if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
461             {
462                 _targetsTreeMethods.Detach(LinksHeader.GetFirstAsTargetPointer(_header),
                    ↪  linkIndex);
463             }
464             Link.SetSource(link, values[Constants.SourcePart]);
465             Link.SetTarget(link, values[Constants.TargetPart]);
466             if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
467             {
468                 _sourcesTreeMethods.Attach(LinksHeader.GetFirstAsSourcePointer(_header),
                    ↪  linkIndex);
469             }
470             if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
471             {
```

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

```
        /// <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 readonly ResizableDirectMemoryLinks<TLink> _memory;
            private readonly LinksCombinedConstants<TLink, TLink, int> _constants;
            protected readonly IntPtr Links;
            protected readonly IntPtr Header;

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

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

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

            public TLink this[TLink index]
            {
                get
                {
                    var root = GetTreeRoot();
                    if (GreaterOrEqualThan(index, GetSize(root)))
```

```
            {
                return GetZero();
            }
            while (!EqualToZero(root))
            {
                var left = GetLeftOrDefault(root);
                var leftSize = GetSizeOrZero(left);
                if (LessThan(index, leftSize))
                {
                    root = left;
                    continue;
                }
                if (IsEquals(index, leftSize))
                {
                    return root;
                }
                root = GetRightOrDefault(root);
                index = Subtract(index, Increment(leftSize));
            }
            return GetZero(); // TODO: Impossible situation exception (only if tree
            ↪  structure broken)
        }
    }

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

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

    public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
    {
        var root = GetTreeRoot();
        if (EqualToZero(root))
        {
            return _constants.Continue;
        }
        TLink first = GetZero(), current = root;
        while (!EqualToZero(current))
        {
            var @base = GetBasePartValue(current);
            if (GreaterOrEqualThan(@base, link))
            {
                if (IsEquals(@base, link))
                {
                    first = current;
```

```
                    }
                    current = GetLeftOrDefault(current);
                }
                else
                {
                    current = GetRightOrDefault(current);
                }
            }
            if (!EqualToZero(first))
            {
                current = first;
                while (true)
                {
                    if (IsEquals(handler(_memory.GetLinkStruct(current)), _constants.Break))
                    {
                        return _constants.Break;
                    }
                    current = GetNext(current);
                    if (EqualToZero(current) || !IsEquals(GetBasePartValue(current), link))
                    {
                        break;
                    }
                }
            }
            return _constants.Continue;
        }

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

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

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

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

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

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

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

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

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

        protected override void SetSize(TLink node, TLink size)
        {
```

```
184                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
185                 (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsSourceOffset).SetValue(Bit.PartialWrite(previousValue, size, 5,
                    ↪ -5));
186             }
187
188             protected override bool GetLeftIsChild(TLink node)
189             {
190                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
191                 return (Integer<TLink>)Bit.PartialRead(previousValue, 4, 1);
192             }
193
194             protected override void SetLeftIsChild(TLink node, bool value)
195             {
196                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
197                 var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 4,
                    ↪ 1);
198                 (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsSourceOffset).SetValue(modified);
199             }
200
201             protected override bool GetRightIsChild(TLink node)
202             {
203                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
204                 return (Integer<TLink>)Bit.PartialRead(previousValue, 3, 1);
205             }
206
207             protected override void SetRightIsChild(TLink node, bool value)
208             {
209                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
210                 var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 3,
                    ↪ 1);
211                 (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsSourceOffset).SetValue(modified);
212             }
213
214             protected override sbyte GetBalance(TLink node)
215             {
216                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
217                 var value = (ulong)(Integer<TLink>)Bit.PartialRead(previousValue, 0, 3);
218                 var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
                    ↪ 124 : value & 3);
219                 return unpackedValue;
220             }
221
222             protected override void SetBalance(TLink node, sbyte value)
223             {
224                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
225                 var packagedValue = (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) | value &
                    ↪ 3);
226                 var modified = Bit.PartialWrite(previousValue, packagedValue, 0, 3);
227                 (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsSourceOffset).SetValue(modified);
228             }
229
230             protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
231             {
232                 var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
                    ↪ Link.SourceOffset).GetValue<TLink>();
233                 var secondSource = (Links.GetElement(LinkSizeInBytes, second) +
                    ↪ Link.SourceOffset).GetValue<TLink>();
234                 return LessThan(firstSource, secondSource) ||
235                        (IsEquals(firstSource, secondSource) &&
                           ↪ LessThan((Links.GetElement(LinkSizeInBytes, first) +
                           ↪ Link.TargetOffset).GetValue<TLink>(),
                           ↪ (Links.GetElement(LinkSizeInBytes, second) +
                           ↪ Link.TargetOffset).GetValue<TLink>()));
236             }
237
```

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

```
295
296             protected override IntPtr GetRightPointer(TLink node) =>
                ↪ Links.GetElement(LinkSizeInBytes, node) + Link.RightAsTargetOffset;
297
298             protected override TLink GetLeftValue(TLink node) =>
                ↪ (Links.GetElement(LinkSizeInBytes, node) +
                ↪ Link.LeftAsTargetOffset).GetValue<TLink>();
299
300             protected override TLink GetRightValue(TLink node) =>
                ↪ (Links.GetElement(LinkSizeInBytes, node) +
                ↪ Link.RightAsTargetOffset).GetValue<TLink>();
301
302             protected override TLink GetSize(TLink node)
303             {
304                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
305                 return Bit.PartialRead(previousValue, 5, -5);
306             }
307
308             protected override void SetLeft(TLink node, TLink left) =>
                ↪ (Links.GetElement(LinkSizeInBytes, node) +
                ↪ Link.LeftAsTargetOffset).SetValue(left);
309
310             protected override void SetRight(TLink node, TLink right) =>
                ↪ (Links.GetElement(LinkSizeInBytes, node) +
                ↪ Link.RightAsTargetOffset).SetValue(right);
311
312             protected override void SetSize(TLink node, TLink size)
313             {
314                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
315                 (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsTargetOffset).SetValue(Bit.PartialWrite(previousValue, size, 5,
                    ↪ -5));
316             }
317
318             protected override bool GetLeftIsChild(TLink node)
319             {
320                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
321                 return (Integer<TLink>)Bit.PartialRead(previousValue, 4, 1);
322             }
323
324             protected override void SetLeftIsChild(TLink node, bool value)
325             {
326                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
327                 var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 4,
                    ↪ 1);
328                 (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsTargetOffset).SetValue(modified);
329             }
330
331             protected override bool GetRightIsChild(TLink node)
332             {
333                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
334                 return (Integer<TLink>)Bit.PartialRead(previousValue, 3, 1);
335             }
336
337             protected override void SetRightIsChild(TLink node, bool value)
338             {
339                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
340                 var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 3,
                    ↪ 1);
341                 (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsTargetOffset).SetValue(modified);
342             }
343
344             protected override sbyte GetBalance(TLink node)
345             {
346                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
347                 var value = (ulong)(Integer<TLink>)Bit.PartialRead(previousValue, 0, 3);
348                 var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
                    ↪ 124 : value & 3);
```

```
                return unpackedValue;
            }

            protected override void SetBalance(TLink node, sbyte value)
            {
                var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
                var packagedValue = (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) | value &
                ↪ 3);
                var modified = Bit.PartialWrite(previousValue, packagedValue, 0, 3);
                (Links.GetElement(LinkSizeInBytes, node) +
                ↪ Link.SizeAsTargetOffset).SetValue(modified);
            }

            protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
            {
                var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
                ↪ Link.TargetOffset).GetValue<TLink>();
                var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
                ↪ Link.TargetOffset).GetValue<TLink>();
                return LessThan(firstTarget, secondTarget) ||
                       (IsEquals(firstTarget, secondTarget) &&
                        ↪ LessThan((Links.GetElement(LinkSizeInBytes, first) +
                        ↪ Link.SourceOffset).GetValue<TLink>(),
                        ↪ (Links.GetElement(LinkSizeInBytes, second) +
                        ↪ Link.SourceOffset).GetValue<TLink>()));
            }

            protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
            {
                var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
                ↪ Link.TargetOffset).GetValue<TLink>();
                var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
                ↪ Link.TargetOffset).GetValue<TLink>();
                return GreaterThan(firstTarget, secondTarget) ||
                       (IsEquals(firstTarget, secondTarget) &&
                        ↪ GreaterThan((Links.GetElement(LinkSizeInBytes, first) +
                        ↪ Link.SourceOffset).GetValue<TLink>(),
                        ↪ (Links.GetElement(LinkSizeInBytes, second) +
                        ↪ Link.SourceOffset).GetValue<TLink>()));
            }

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

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

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

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Disposables;
using Platform.Collections.Arrays;
using Platform.Singletons;
using Platform.Memory;
using Platform.Data.Exceptions;
using Platform.Data.Constants;

#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.
```

```
28          /// Так как во вне не обязательно будет доступен unsafe C#.
29          /// </remarks>
30          public static readonly int LinkSizeInBytes = sizeof(Link);
31
32          public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
33
34          private struct Link
35          {
36              public id Source;
37              public id Target;
38              public id LeftAsSource;
39              public id RightAsSource;
40              public id SizeAsSource;
41              public id LeftAsTarget;
42              public id RightAsTarget;
43              public id SizeAsTarget;
44          }
45
46          private struct LinksHeader
47          {
48              public id AllocatedLinks;
49              public id ReservedLinks;
50              public id FreeLinks;
51              public id FirstFreeLink;
52              public id FirstAsSource;
53              public id FirstAsTarget;
54              public id LastFreeLink;
55              public id Reserved8;
56          }
57
58          private readonly long _memoryReservationStep;
59
60          private readonly IResizableDirectMemory _memory;
61          private LinksHeader* _header;
62          private Link* _links;
63
64          private LinksTargetsTreeMethods _targetsTreeMethods;
65          private LinksSourcesTreeMethods _sourcesTreeMethods;
66
67          // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
            ↪  нужно использовать не список а дерево, так как так можно быстрее проверить на
            ↪  наличие связи внутри
68          private UnusedLinksListMethods _unusedLinksListMethods;
69
70          /// <summary>
71          /// Возвращает общее число связей находящихся в хранилище.
72          /// </summary>
73          private id Total => _header->AllocatedLinks - _header->FreeLinks;
74
75          // TODO: Дать возможность переопределять в конструкторе
76          public LinksCombinedConstants<id, id, int> Constants { get; }
77
78          public UInt64ResizableDirectMemoryLinks(string address) : this(address,
            ↪  DefaultLinksSizeStep) { }
79
80          /// <summary>
81          /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
            ↪  минимальным шагом расширения базы данных.
82          /// </summary>
83          /// <param name="address">Полный пусть к файлу базы данных.</param>
84          /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
            ↪  байтах.</param>
85          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
```

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

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public id Each(Func<IList<id>, id> handler, IList<id> restrictions)
    {
        if (restrictions.Count == 0)
        {
            for (id link = 1; link <= _header->AllocatedLinks; link++)
            {
                if (Exists(link))
                {
                    if (handler(GetLinkStruct(link)) == Constants.Break)
                    {
                        return Constants.Break;
                    }
                }
            }
            return Constants.Continue;
        }
        if (restrictions.Count == 1)
        {
            var index = restrictions[Constants.IndexPart];
            if (index == Constants.Any)
            {
                return Each(handler, ArrayPool<ulong>.Empty);
            }
            if (!Exists(index))
            {
                return Constants.Continue;
            }
            return handler(GetLinkStruct(index));
        }
        if (restrictions.Count == 2)
        {
            var index = restrictions[Constants.IndexPart];
            var value = restrictions[1];
```

```
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;
315                     }
316                     if (source == Constants.Any && target == Constants.Any)
317                     {
318                         return handler(GetLinkStruct(index));
319                     }
320                     var storedLinkValue = GetLinkUnsafe(index);
321                     if (source != Constants.Any && target != Constants.Any)
322                     {
323                         if (storedLinkValue->Source == source &&
324                             storedLinkValue->Target == target)
325                         {
326                             return handler(GetLinkStruct(index));
327                         }
328                         return Constants.Continue;
329                     }
330                     var value = default(id);
```

```
                    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];

```

```
406         /// <remarks>
407         /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
            ↪  пространство
408         /// </remarks>
409         public id Create()
410         {
411             var freeLink = _header->FirstFreeLink;
412             if (freeLink != Constants.Null)
413             {
414                 _unusedLinksListMethods.Detach(freeLink);
415             }
416             else
417             {
418                 if (_header->AllocatedLinks > Constants.MaxPossibleIndex)
419                 {
420                     throw new LinksLimitReachedException(Constants.MaxPossibleIndex);
421                 }
422                 if (_header->AllocatedLinks >= _header->ReservedLinks - 1)
423                 {
424                     _memory.ReservedCapacity += _memoryReservationStep;
425                     SetPointers(_memory);
426                     _header->ReservedLinks = (id)(_memory.ReservedCapacity / sizeof(Link));
427                 }
428                 _header->AllocatedLinks++;
429                 _memory.UsedCapacity += sizeof(Link);
430                 freeLink = _header->AllocatedLinks;
431             }
432             return freeLink;
433         }
434 
435         public void Delete(id link)
436         {
437             if (link < _header->AllocatedLinks)
438             {
439                 _unusedLinksListMethods.AttachAsFirst(link);
440             }
441             else if (link == _header->AllocatedLinks)
442             {
443                 _header->AllocatedLinks--;
444                 _memory.UsedCapacity -= sizeof(Link);
445                 // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
                    ↪  пока не дойдём до первой существующей связи
446                 // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
447                 while (_header->AllocatedLinks > 0 && IsUnusedLink(_header->AllocatedLinks))
448                 {
449                     _unusedLinksListMethods.Detach(_header->AllocatedLinks);
450                     _header->AllocatedLinks--;
451                     _memory.UsedCapacity -= sizeof(Link);
452                 }
453             }
454         }
455 
456         /// <remarks>
457         /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
            ↪  адрес реально поменялся
458         ///
459         /// Указатель this.links может быть в том же месте,
460         /// так как 0-я связь не используется и имеет такой же размер как Header,
461         /// поэтому header размещается в том же месте, что и 0-я связь
462         /// </remarks>
463         private void SetPointers(IResizableDirectMemory memory)
464         {
465             if (memory == null)
466             {
467                 _header = null;
468                 _links = null;
469                 _unusedLinksListMethods = null;
470                 _targetsTreeMethods = null;
471                 _unusedLinksListMethods = null;
472             }
473             else
474             {
475                 _header = (LinksHeader*)(void*)memory.Pointer;
476                 _links = (Link*)(void*)memory.Pointer;
477                 _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
478                 _targetsTreeMethods = new LinksTargetsTreeMethods(this);
479                 _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
480             }
481         }
```

```

        [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;
            var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 | 124 : value & 3);
            return unpackedValue;
        }

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

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

```
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               {
289                   Links[node].LeftAsSource = 0UL;
290                   Links[node].RightAsSource = 0UL;
291                   Links[node].SizeAsSource = 0UL;
292               }
293
294               [MethodImpl(MethodImplOptions.AggressiveInlining)]
295               protected override ulong GetZero() => 0UL;
296
297               [MethodImpl(MethodImplOptions.AggressiveInlining)]
298               protected override ulong GetOne() => 1UL;
299
300               [MethodImpl(MethodImplOptions.AggressiveInlining)]
301               protected override ulong GetTwo() => 2UL;
302
303               [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
304             protected override bool ValueEqualToZero(IntPtr pointer) =>
                ↪  *(ulong*)pointer.ToPointer() == 0UL;
305
306             [MethodImpl(MethodImplOptions.AggressiveInlining)]
307             protected override bool EqualToZero(ulong value) => value == 0UL;
308
309             [MethodImpl(MethodImplOptions.AggressiveInlining)]
310             protected override bool IsEquals(ulong first, ulong second) => first == second;
311
312             [MethodImpl(MethodImplOptions.AggressiveInlining)]
313             protected override bool GreaterThanZero(ulong value) => value > 0UL;
314
315             [MethodImpl(MethodImplOptions.AggressiveInlining)]
316             protected override bool GreaterThan(ulong first, ulong second) => first > second;
317
318             [MethodImpl(MethodImplOptions.AggressiveInlining)]
319             protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >=
                ↪  second;
320
321             [MethodImpl(MethodImplOptions.AggressiveInlining)]
322             protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0
                ↪  is always true for ulong
323
324             [MethodImpl(MethodImplOptions.AggressiveInlining)]
325             protected override bool LessOrEqualThanZero(ulong value) => value == 0; // value is
                ↪  always >= 0 for ulong
326
327             [MethodImpl(MethodImplOptions.AggressiveInlining)]
328             protected override bool LessOrEqualThan(ulong first, ulong second) => first <=
                ↪  second;
329
330             [MethodImpl(MethodImplOptions.AggressiveInlining)]
331             protected override bool LessThanZero(ulong value) => false; // value < 0 is always
                ↪  false for ulong
332
333             [MethodImpl(MethodImplOptions.AggressiveInlining)]
334             protected override bool LessThan(ulong first, ulong second) => first < second;
335
336             [MethodImpl(MethodImplOptions.AggressiveInlining)]
337             protected override ulong Increment(ulong value) => ++value;
338
339             [MethodImpl(MethodImplOptions.AggressiveInlining)]
340             protected override ulong Decrement(ulong value) => --value;
341
342             [MethodImpl(MethodImplOptions.AggressiveInlining)]
343             protected override ulong Add(ulong first, ulong second) => first + second;
344
345             [MethodImpl(MethodImplOptions.AggressiveInlining)]
346             protected override ulong Subtract(ulong first, ulong second) => first - second;
347         }
348
349         private class LinksTargetsTreeMethods : LinksTreeMethodsBase
350         {
351             public LinksTargetsTreeMethods(UInt64ResizableDirectMemoryLinks memory)
352                 : base(memory)
353             {
354             }
355
356             //protected override IntPtr GetLeft(ulong node) => new
                ↪  IntPtr(&Links[node].LeftAsTarget);
357
358             //protected override IntPtr GetRight(ulong node) => new
                ↪  IntPtr(&Links[node].RightAsTarget);
359
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);
431                 var modified = (previousValue & 4294967287) | ((value ? 1UL : 0UL) << 3);
432                 Links[node].SizeAsTarget = modified;
433             }
434
435             protected override sbyte GetBalance(ulong node)
436             {
437                 var previousValue = Links[node].SizeAsTarget;
438                 //var value = Math.PartialRead(previousValue, 0, 3);
439                 var value = previousValue & 7;
440                 var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
                    ↪  124 : value & 3);
441                 return unpackedValue;
442             }
443
```

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
        {
            var frequency = data.Frequency;
```

```
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             {
24                 return sequence[0];
25             }
26             var links = Links;
27             if (length == 2)
28             {
29                 return links.GetOrCreate(sequence[0], sequence[1]);
30             }
31             sequence = sequence.ToArray();
32             var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
33             while (length > 2)
34             {
35                 var levelRepeat = 1;
36                 var currentLevel = levels[0];
37                 var previousLevel = levels[0];
38                 var skipOnce = false;
```

```
                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]);
            }
        }

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

        public void PrintFrequencies(IList<TLink> sequence)
        {
            for (var i = 1; i < sequence.Count; i++)
```

```
            {
                PrintFrequency(sequence[i - 1], sequence[i]);
            }
        }

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

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

        public void ValidateFrequencies()
        {
            foreach (var entry in _doubletsCache)
            {
                var value = entry.Value;
                var linkIndex = value.Link;
                if (!_equalityComparer.Equals(linkIndex, default))
                {
                    var frequency = value.Frequency;
                    var count = _frequencyCounter.Count(linkIndex);
                    // TODO: Why `frequency` always greater than `count` by 1?
                    if (((_comparer.Compare(frequency, count) > 0) &&
                    ↪  (_comparer.Compare(Arithmetic.Subtract(frequency, count),
                    ↪  Integer<TLink>.One) > 0))
                     || ((_comparer.Compare(count, frequency) > 0) &&
                     ↪  (_comparer.Compare(Arithmetic.Subtract(count, frequency),
                     ↪  Integer<TLink>.One) > 0)))
                    {
                        throw new InvalidOperationException("Frequencies validation failed.");
                    }
                }
                //else
                //{
                //    if (value.Frequency > 0)
                //    {
                //        var frequency = value.Frequency;
                //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
                //        var count = _countLinkFrequency(linkIndex);

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

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

```
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;
        }

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

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

```
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.Counters
{
```

```
    public class TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
    ↪ TotalSequenceSymbolFrequencyOneOffCounter<TLink>
    {
        private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;

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

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

./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.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 TotalSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
    {
        private readonly ILinks<TLink> _links;
        public TotalSequenceSymbolFrequencyCounter(ILinks<TLink> links) => _links = links;
        public TLink Count(TLink symbol) => new
        ↪ TotalSequenceSymbolFrequencyOneOffCounter<TLink>(_links, symbol).Count();
    }
}
```

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

```
using System.Collections.Generic;
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.Counters
{
    public class TotalSequenceSymbolFrequencyOneOffCounter<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 _symbol;
        protected readonly HashSet<TLink> _visits;
        protected TLink _total;

        public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
        {
            _links = links;
            _symbol = symbol;
            _visits = new HashSet<TLink>();
            _total = default;
        }

        public TLink Count()
        {
            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;
        }

```

```
45          private void Increment(TLink link)
46          {
47              var previousFrequency = _frequencyPropertyOperator.Get(link);
48              var frequency = _frequencyIncrementer.Increment(previousFrequency);
49              _frequencyPropertyOperator.Set(link, frequency);
50          }
51      }
52  }
```

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

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

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

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

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

```
1   using System.Collections.Generic;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Data.Doublets.Sequences.Indexes
6   {
7       public class SynchronizedSequenceIndex<TLink> : ISequenceIndex<TLink>
8       {
```

```
        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.
    /// </summary>
    /// <remarks>
    /// Обязательно реализовать атомарность каждого публичного метода.
    ///
    /// TODO:
    ///
    /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
    /// через естественную группировку по unicode типам, все whitespace вместе, все символы
    ↪  вместе, все числа вместе и т.п.
    /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
    ↪  графа)
    ///
```

```
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;
61          public readonly SynchronizedLinks<ulong> Links;
62          public 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             {
```

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

        #region Count

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

        private ulong CountUsages(params ulong[] restrictions)
        {
            if (restrictions.Length == 0)
            {
                return 0;
            }
            if (restrictions.Length == 1) // Первая связь это адрес
            {
                if (restrictions[0] == _constants.Null)
                {
                    return 0;
                }
                if (Options.UseSequenceMarker)
                {
                    var elementsLink = GetSequenceElements(restrictions[0]);
                    var sequenceLink = GetSequenceByElements(elementsLink);
                    if (sequenceLink != _constants.Null)
                    {
                        return Links.Count(sequenceLink) + Links.Count(elementsLink) - 1;
                    }
                    return Links.Count(elementsLink);
                }
                return Links.Count(restrictions[0]);
            }
            throw new NotImplementedException();
        }

        #endregion

        #region Create

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

```
184                 Links.EnsureEachLinkExists(sequence);
185                 return CreateCore(sequence);
186             });
187         }
188 
189         private ulong CreateCore(params ulong[] sequence)
190         {
191             if (Options.UseIndex)
192             {
193                 Options.Index.Add(sequence);
194             }
195             var sequenceRoot = default(ulong);
196             if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
197             {
198                 var matches = Each(sequence);
199                 if (matches.Count > 0)
200                 {
201                     sequenceRoot = matches[0];
202                 }
203             }
204             else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
205             {
206                 return CompactCore(sequence);
207             }
208             if (sequenceRoot == default)
209             {
210                 sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
211             }
212             if (Options.UseSequenceMarker)
213             {
214                 Links.Unsync.CreateAndUpdate(Options.SequenceMarkerLink, sequenceRoot);
215             }
216             return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
217         }
218 
219         #endregion
220 
221         #region Each
222 
223         public List<ulong> Each(params ulong[] sequence)
224         {
225             var results = new List<ulong>();
226             Each(results.AddAndReturnTrue, sequence);
227             return results;
228         }
229 
230         public bool Each(Func<ulong, bool> handler, IList<ulong> sequence)
231         {
232             return Sync.ExecuteReadOperation(() =>
233             {
234                 if (sequence.IsNullOrEmpty())
235                 {
236                     return true;
237                 }
238                 Links.EnsureEachLinkIsAnyOrExists(sequence);
239                 if (sequence.Count == 1)
240                 {
241                     var link = sequence[0];
242                     if (link == _constants.Any)
243                     {
244                         return Links.Unsync.Each(_constants.Any, _constants.Any, handler);
245                     }
246                     return handler(link);
247                 }
248                 if (sequence.Count == 2)
249                 {
250                     return Links.Unsync.Each(sequence[0], sequence[1], handler);
251                 }
252                 if (Options.UseIndex && !Options.Index.MightContain(sequence))
253                 {
254                     return false;
255                 }
256                 return EachCore(handler, sequence);
257             });
258         }
259 
260         private bool EachCore(Func<ulong, bool> handler, IList<ulong> sequence)
261         {
262             var matcher = new Matcher(this, sequence, new HashSet<LinkIndex>(), handler);
```

```
263             // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
                ↪ Id.
264             Func<ulong, bool> innerHandler = Options.UseSequenceMarker ? (Func<ulong,
                ↪ bool>)matcher.HandleFullMatchedSequence : matcher.HandleFullMatched;
265             //if (sequence.Length >= 2)
266             if (!StepRight(innerHandler, sequence[0], sequence[1]))
267             {
268                 return false;
269             }
270             var last = sequence.Count - 2;
271             for (var i = 1; i < last; i++)
272             {
273                 if (!PartialStepRight(innerHandler, sequence[i], sequence[i + 1]))
274                 {
275                     return false;
276                 }
277             }
278             if (sequence.Count >= 3)
279             {
280                 if (!StepLeft(innerHandler, sequence[sequence.Count - 2],
                    ↪ sequence[sequence.Count - 1]))
281                 {
282                     return false;
283                 }
284             }
285             return true;
286         }
287
288         private bool PartialStepRight(Func<ulong, bool> handler, ulong left, ulong right)
289         {
290             return Links.Unsync.Each(_constants.Any, left, doublet =>
291             {
292                 if (!StepRight(handler, doublet, right))
293                 {
294                     return false;
295                 }
296                 if (left != doublet)
297                 {
298                     return PartialStepRight(handler, doublet, right);
299                 }
300                 return true;
301             });
302         }
303
304         private bool StepRight(Func<ulong, bool> handler, ulong left, ulong right) =>
            ↪ Links.Unsync.Each(left, _constants.Any, rightStep => TryStepRightUp(handler, right,
            ↪ rightStep));
305
306         private bool TryStepRightUp(Func<ulong, bool> handler, ulong right, ulong stepFrom)
307         {
308             var upStep = stepFrom;
309             var firstSource = Links.Unsync.GetTarget(upStep);
310             while (firstSource != right && firstSource != upStep)
311             {
312                 upStep = firstSource;
313                 firstSource = Links.Unsync.GetSource(upStep);
314             }
315             if (firstSource == right)
316             {
317                 return handler(stepFrom);
318             }
319             return true;
320         }
321
322         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);
399                 var newSequenceLink = GetSequenceByElements(newSequenceElements);
400                 if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
401                 {
402                     if (sequenceLink != _constants.Null)
403                     {
404                         Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
405                     }
406                     Links.Unsync.MergeUsages(sequenceElements, newSequenceElements);
407                 }
408                 ClearGarbage(sequenceElementsContents.Source);
409                 ClearGarbage(sequenceElementsContents.Target);
410             }
```

```
            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;
            foreach (var part in Options.Walker.Walk(sequence))
            {
                if (!handler(part))
                {
                    return false;
                }
            }
            return true;
        });
    }

    public class Matcher : RightSequenceWalker<ulong>
    {
```

```
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             {
707                 foreach (var sequenceToMatch in sequencesToMatch)
708                 {
709                     if (PartialMatch(sequenceToMatch))
710                     {
711                         _results.Add(sequenceToMatch);
712                     }
713                 }
714             }
715 
716             public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<ulong>
    ↪  sequencesToMatch)
717             {
718                 foreach (var sequenceToMatch in sequencesToMatch)
```

```
                {
                    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)Numbers.Math.Catalan(stopAt - startAt)];
            var last = 0;
            for (var splitter = startAt; splitter < stopAt; splitter++)
            {
                var left = CreateAllVariants2Core(sequence, startAt, splitter);
```

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

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

    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;
        }

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

```
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);
282                 }
283                 return true;
284             });
285         }
286
287         private void StepRight(Action<ulong> handler, ulong left, ulong right)
288         {
289             Links.Unsync.Each(left, _constants.Any, rightStep =>
290             {
291                 TryStepRightUp(handler, right, rightStep);
292                 return true;
293             });
294         }
295
296         private void TryStepRightUp(Action<ulong> handler, ulong right, ulong stepFrom)
```

```
297             {
298                 var upStep = stepFrom;
299                 var firstSource = Links.Unsync.GetTarget(upStep);
300                 while (firstSource != right && firstSource != upStep)
301                 {
302                     upStep = firstSource;
303                     firstSource = Links.Unsync.GetSource(upStep);
304                 }
305                 if (firstSource == right)
306                 {
307                     handler(stepFrom);
308                 }
309             }
310 
311             // TODO: Test
312             private void PartialStepLeft(Action<ulong> handler, ulong left, ulong right)
313             {
314                 Links.Unsync.Each(right, _constants.Any, doublet =>
315                 {
316                     StepLeft(handler, left, doublet);
317                     if (right != doublet)
318                     {
319                         PartialStepLeft(handler, left, doublet);
320                     }
321                     return true;
322                 });
323             }
324 
325             private void StepLeft(Action<ulong> handler, ulong left, ulong right)
326             {
327                 Links.Unsync.Each(_constants.Any, right, leftStep =>
328                 {
329                     TryStepLeftUp(handler, left, leftStep);
330                     return true;
331                 });
332             }
333 
334             private void TryStepLeftUp(Action<ulong> handler, ulong left, ulong stepFrom)
335             {
336                 var upStep = stepFrom;
337                 var firstTarget = Links.Unsync.GetSource(upStep);
338                 while (firstTarget != left && firstTarget != upStep)
339                 {
340                     upStep = firstTarget;
341                     firstTarget = Links.Unsync.GetTarget(upStep);
342                 }
343                 if (firstTarget == left)
344                 {
345                     handler(stepFrom);
346                 }
347             }
348 
349             private bool StartsWith(ulong sequence, ulong link)
350             {
351                 var upStep = sequence;
352                 var firstSource = Links.Unsync.GetSource(upStep);
353                 while (firstSource != link && firstSource != upStep)
354                 {
355                     upStep = firstSource;
356                     firstSource = Links.Unsync.GetSource(upStep);
357                 }
358                 return firstSource == link;
359             }
360 
361             private bool EndsWith(ulong sequence, ulong link)
362             {
363                 var upStep = sequence;
364                 var lastTarget = Links.Unsync.GetTarget(upStep);
365                 while (lastTarget != link && lastTarget != upStep)
366                 {
367                     upStep = lastTarget;
368                     lastTarget = Links.Unsync.GetTarget(upStep);
369                 }
370                 return lastTarget == link;
371             }
372 
373             public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
374             {
375                 return Sync.ExecuteReadOperation(() =>
```

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

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

        public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
        {
            return Sync.ExecuteReadOperation(() =>
            {
                var results = new HashSet<ulong>();
                if (sequence.Length > 0)
                {
                    Links.EnsureEachLinkExists(sequence);
                    var firstElement = sequence[0];
                    if (sequence.Length == 1)
                    {
```

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

    public const int MaxSequenceFormatSize = 200;

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

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

    private string FormatSequence(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.IsPartialPoint(x), element => //
                ↪  entered.AddAndReturnVoid, x => { }, entered.DoNotContains
                {
                    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('}');
```

```
520             return sb.ToString();
521         }
522 
523         public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
            ↪ knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
            ↪ knownElements);
524 
525         public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
            ↪ LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
            ↪ Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
            ↪ sequenceLink, elementToString, insertComma, knownElements));
526 
527         private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
            ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
            ↪ LinkIndex[] knownElements)
528         {
529             var linksInSequence = new HashSet<ulong>(knownElements);
530             var entered = new HashSet<ulong>();
531             var sb = new StringBuilder();
532             sb.Append('{');
533             if (links.Exists(sequenceLink))
534             {
535                 StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
536                     x => linksInSequence.Contains(x) || links.IsFullPoint(x),
                        ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
537                     {
538                         if (insertComma && sb.Length > 1)
539                         {
540                             sb.Append(',');
541                         }
542                         if (entered.Contains(element))
543                         {
544                             sb.Append('{');
545                             elementToString(sb, element);
546                             sb.Append('}');
547                         }
548                         else
549                         {
550                             elementToString(sb, element);
551                         }
552                         if (sb.Length < MaxSequenceFormatSize)
553                         {
554                             return true;
555                         }
556                         sb.Append(insertComma ? ", ..." : "...");
557                         return false;
558                     });
559             }
560             sb.Append('}');
561             return sb.ToString();
562         }
563 
564         public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
565         {
566             return Sync.ExecuteReadOperation(() =>
567             {
568                 if (sequence.Length > 0)
569                 {
570                     Links.EnsureEachLinkExists(sequence);
571                     var results = new HashSet<ulong>();
572                     for (var i = 0; i < sequence.Length; i++)
573                     {
574                         AllUsagesCore(sequence[i], results);
575                     }
576                     var filteredResults = new List<ulong>();
577                     var linksInSequence = new HashSet<ulong>(sequence);
578                     foreach (var result in results)
579                     {
580                         var filterPosition = -1;
581                         StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
                            ↪ Links.Unsync.GetTarget,
582                             x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
                                ↪ x =>
583                             {
584                                 if (filterPosition == (sequence.Length - 1))
585                                 {
586                                     return false;
587                                 }
```

```
                            if (filterPosition >= 0)
                            {
                                if (x == sequence[filterPosition + 1])
                                {
                                    filterPosition++;
                                }
                                else
                                {
                                    return false;
                                }
                            }
                            if (filterPosition < 0)
                            {
                                if (x == sequence[0])
                                {
                                    filterPosition = 0;
                                }
                            }
                            return true;
                        });
                    if (filterPosition == (sequence.Length - 1))
                    {
                        filteredResults.Add(result);
                    }
                }
                return filteredResults;
            }
            return new List<ulong>();
        });
    }

    public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
    {
        return Sync.ExecuteReadOperation(() =>
        {
            if (sequence.Length > 0)
            {
                Links.EnsureEachLinkExists(sequence);
                var results = new HashSet<ulong>();
                for (var i = 0; i < sequence.Length; i++)
                {
                    AllUsagesCore(sequence[i], results);
                }
                var filteredResults = new HashSet<ulong>();
                var matcher = new Matcher(this, sequence, filteredResults, null);
                matcher.AddAllPartialMatchedToResults(results);
                return filteredResults;
            }
            return new HashSet<ulong>();
        });
    }

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

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

    //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
    //{
```

```
        //    return Sync.ExecuteReadOperation(() =>
        //    {
        //        if (sequence.Length > 0)
        //        {
        //            _links.EnsureEachLinkIsAnyOrExists(sequence);

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

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

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

        //            firstResults.IntersectWith(lastResults);

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

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

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

        public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
        {
            return Sync.ExecuteReadOperation(() =>
            {
                if (sequence.Length > 0)
                {
                    Links.EnsureEachLinkIsAnyOrExists(sequence);
                    var firstResults = new HashSet<ulong>();
                    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();
            }
        }

        private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<ulong, bool>
        ↪  outerHandler)
        {
            bool handler(ulong doublet)
            {
                if (usages.Add(doublet))
                {
                    if (!outerHandler(doublet))
                    {
                        return false;
                    }
                    if (!AllUsagesCore1(doublet, usages, outerHandler))
                    {
                        return false;
```

```
                }
            }
            return true;
        }
        return Links.Unsync.Each(link, _constants.Any, handler)
            && Links.Unsync.Each(_constants.Any, link, handler);
    }

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

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

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

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

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

        private bool CalculateCore(ulong link)
        {
            if (_totals[link] == 0)
            {
                var total = 1UL;
                _totals[link] = total;
                var visitedChildren = new HashSet<ulong>();
                bool linkCalculator(ulong child)
                {
                    if (link != child && visitedChildren.Add(child))
                    {
                        total += _totals[child] == 0 ? 1 : _totals[child];
                    }
                    return true;
                }
                _links.Unsync.Each(link, _constants.Any, linkCalculator);
                _links.Unsync.Each(_constants.Any, link, linkCalculator);
                _totals[link] = total;
            }
            return true;
        }
    }

    private class AllUsagesCalculator2
    {
        private readonly SynchronizedLinks<ulong> _links;
        private readonly ulong[] _totals;

        public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
        {
            _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
        {
            private readonly ILinks<ulong> _links;
            private readonly HashSet<ulong> _usages;

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

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

```
                    _links.Each(_constants.Any, link, Collect);
                }
                return true;
            }
        }

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

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

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

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

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

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

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

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

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

```
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                {
```

```
                throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                ↪ $"patternSequence[{i}]");
            }
        }
    }

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

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

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

```
                    next.Clear();
                }
            }
            return results;
        });
    }

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

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

```
1643            }
1644
1645            #endregion
1646
1647            #region Walkers
1648
1649            public class PatternMatcher : RightSequenceWalker<ulong>
1650            {
1651                private readonly Sequences _sequences;
1652                private readonly ulong[] _patternSequence;
1653                private readonly HashSet<LinkIndex> _linksInSequence;
1654                private readonly HashSet<LinkIndex> _results;
1655
1656                #region Pattern Match
1657
1658                enum PatternBlockType
1659                {
1660                    Undefined,
1661                    Gap,
1662                    Elements
1663                }
1664
1665                struct PatternBlock
1666                {
1667                    public PatternBlockType Type;
1668                    public long Start;
1669                    public long Stop;
1670                }
1671
1672                private readonly List<PatternBlock> _pattern;
1673                private int _patternPosition;
1674                private long _sequencePosition;
1675
1676                #endregion
1677
1678                public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
                    ↪  HashSet<LinkIndex> results)
1679                    : base(sequences.Links.Unsync, new DefaultStack<ulong>())
1680                {
1681                    _sequences = sequences;
1682                    _patternSequence = patternSequence;
1683                    _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                        ↪  _constants.Any && x != ZeroOrMany));
1684                    _results = results;
1685                    _pattern = CreateDetailedPattern();
1686                }
1687
1688                protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
                    ↪  base.IsElement(link);
1689
1690                public bool PatternMatch(LinkIndex sequenceToMatch)
1691                {
1692                    _patternPosition = 0;
1693                    _sequencePosition = 0;
1694                    foreach (var part in Walk(sequenceToMatch))
1695                    {
1696                        if (!PatternMatchCore(part))
1697                        {
1698                            break;
1699                        }
1700                    }
1701                    return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
                        ↪  - 1 && _pattern[_patternPosition].Start == 0);
1702                }
1703
1704                private List<PatternBlock> CreateDetailedPattern()
1705                {
1706                    var pattern = new List<PatternBlock>();
1707                    var patternBlock = new PatternBlock();
1708                    for (var i = 0; i < _patternSequence.Length; i++)
1709                    {
1710                        if (patternBlock.Type == PatternBlockType.Undefined)
1711                        {
1712                            if (_patternSequence[i] == _constants.Any)
1713                            {
1714                                patternBlock.Type = PatternBlockType.Gap;
1715                                patternBlock.Start = 1;
1716                                patternBlock.Stop = 1;
1717                            }
1718                            else if (_patternSequence[i] == ZeroOrMany)
1719                            {
```

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

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

```

```
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                        {
1863                            _patternPosition++;
1864                            _sequencePosition = 0;
1865                            return false; // Полное соответствие
1866                        }
1867                        else
1868                        {
1869                            if (_sequencePosition > currentPatternBlock.Stop)
1870                            {
1871                                return false; // Соответствие невозможно
1872                            }
1873                            var nextPatternBlock = _pattern[_patternPosition + 1];
1874                            if (_patternSequence[nextPatternBlock.Start] == element)
1875                            {
```

```
                        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; }
        public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting { get; set; }
        public bool EnforceSingleSequenceVersionOnWriteBasedOnNew { get; set; }

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

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

```

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

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

```
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.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) : base(links, stack)
        ↪ { }

        [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);
```

```
25
26          public TLink[] ToArray(TLink sequence)
27          {
28              var length = 1;
29              var array = new TLink[length];
30              array[0] = sequence;
31              if (_isElement(sequence))
32              {
33                  return array;
34              }
35              bool hasElements;
36              do
37              {
38                  length *= 2;
39  #if USEARRAYPOOL
40                  var nextArray = ArrayPool.Allocate<ulong>(length);
41  #else
42                  var nextArray = new TLink[length];
43  #endif
44                  hasElements = false;
45                  for (var i = 0; i < array.Length; i++)
46                  {
47                      var candidate = array[i];
48                      if (_equalityComparer.Equals(array[i], default))
49                      {
50                          continue;
51                      }
52                      var doubletOffset = i * 2;
53                      if (_isElement(candidate))
54                      {
55                          nextArray[doubletOffset] = candidate;
56                      }
57                      else
58                      {
59                          var link = Links.GetLink(candidate);
60                          var linkSource = Links.GetSource(link);
61                          var linkTarget = Links.GetTarget(link);
62                          nextArray[doubletOffset] = linkSource;
63                          nextArray[doubletOffset + 1] = linkTarget;
64                          if (!hasElements)
65                          {
66                              hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
67                          }
68                      }
69                  }
70  #if USEARRAYPOOL
71                  if (array.Length > 1)
72                  {
73                      ArrayPool.Free(array);
74                  }
75  #endif
76                  array = nextArray;
77              }
78              while (hasElements);
79              var filledElementsCount = CountFilledElements(array);
80              if (filledElementsCount == array.Length)
81              {
82                  return array;
83              }
84              else
85              {
86                  return CopyFilledElements(array, filledElementsCount);
87              }
88          }
89
90          [MethodImpl(MethodImplOptions.AggressiveInlining)]
91          private static TLink[] CopyFilledElements(TLink[] array, int filledElementsCount)
92          {
93              var finalArray = new TLink[filledElementsCount];
94              for (int i = 0, j = 0; i < array.Length; i++)
95              {
96                  if (!_equalityComparer.Equals(array[i], default))
97                  {
98                      finalArray[j] = array[i];
99                      j++;
100                 }
101             }
102 #if USEARRAYPOOL
103                 ArrayPool.Free(array);
```

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

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

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

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

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

        [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.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;

        protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : base(links) =>
        ↪  _stack = stack;

        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) => Links.IsPartialPoint(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; }
        public ILinks<T> Sync { get; }
        public ILinks<T> Unsync { get; }

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

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

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

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

```
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          {
38              Index = values.Count > _constants.IndexPart ? values[_constants.IndexPart] :
                ↪  _constants.Null;
39              Source = values.Count > _constants.SourcePart ? values[_constants.SourcePart] :
                ↪  _constants.Null;
40              Target = values.Count > _constants.TargetPart ? values[_constants.TargetPart] :
                ↪  _constants.Null;
41          }
42
43          public UInt64Link(ulong index, ulong source, ulong target)
44          {
45              Index = index;
46              Source = source;
47              Target = target;
48          }
49
50          public UInt64Link(ulong source, ulong target)
51              : this(_constants.Null, source, target)
52          {
53              Source = source;
54              Target = target;
55          }
56
57          public static UInt64Link Create(ulong source, ulong target) => new UInt64Link(source,
            ↪  target);
58
59          public override int GetHashCode() => (Index, Source, Target).GetHashCode();
60
61          public bool IsNull() => Index == _constants.Null
62                                  && Source == _constants.Null
63                                  && Target == _constants.Null;
64
```

```
65          public override bool Equals(object other) => other is UInt64Link &&
            ↪  Equals((UInt64Link)other);
66
67          public bool Equals(UInt64Link other) => Index == other.Index
68                                                  && Source == other.Source
69                                                  && Target == other.Target;
70
71          public static string ToString(ulong index, ulong source, ulong target) => $"({index}:
            ↪  {source}->{target})";
72
73          public static string ToString(ulong source, ulong target) => $"({source}->{target})";
74
75          public static implicit operator ulong[](UInt64Link link) => link.ToArray();
76
77          public static implicit operator UInt64Link(ulong[] linkArray) => new
            ↪  UInt64Link(linkArray);
78
79          public override string ToString() => Index == _constants.Null ? ToString(Source, Target)
            ↪  : ToString(Index, Source, Target);
80
81          #region IList
82
83          public ulong this[int index]
84          {
85              get
86              {
87                  Ensure.Always.ArgumentInRange(index, new Range<int>(0, Length - 1),
                    ↪  nameof(index));
88                  if (index == _constants.IndexPart)
89                  {
90                      return Index;
91                  }
92                  if (index == _constants.SourcePart)
93                  {
94                      return Source;
95                  }
96                  if (index == _constants.TargetPart)
97                  {
98                      return Target;
99                  }
100                 throw new NotSupportedException(); // Impossible path due to
                    ↪  Ensure.ArgumentInRange
101             }
102             set => throw new NotSupportedException();
103         }
104
105         public int Count => Length;
106
107         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.Always.ArgumentNotNull(array, nameof(array));
127             Ensure.Always.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
                ↪  nameof(arrayIndex));
128             if (arrayIndex + Length > array.Length)
129             {
130                 throw new ArgumentException();
131             }
132             array[arrayIndex++] = Index;
133             array[arrayIndex++] = Source;
134             array[arrayIndex] = Target;
135         }
136
```

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

        public int IndexOf(ulong item)
        {
            if (Index == item)
            {
                return _constants.IndexPart;
            }
            if (Source == item)
            {
                return _constants.SourcePart;
            }
            if (Target == item)
            {
                return _constants.TargetPart;
            }

            return -1;
        }

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

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

        #endregion
    }
}
```

./Platform.Data.Doublets/UInt64LinkExtensions.cs

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

namespace Platform.Data.Doublets
{
    public static class UInt64LinkExtensions
    {
        public static bool IsFullPoint(this UInt64Link link) => Point<ulong>.IsFullPoint(link);
        public static bool IsPartialPoint(this UInt64Link link) =>
        ↪ Point<ulong>.IsPartialPoint(link);
    }
}
```

./Platform.Data.Doublets/UInt64LinksExtensions.cs

```
using System;
using System.Text;
using System.Collections.Generic;
using Platform.Singletons;
using Platform.Data.Constants;
using Platform.Data.Exceptions;
using Platform.Data.Doublets.Unicode;

#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>();
            links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
            ↪ renderDebug);
            return sb.ToString();
        }

        public static void AppendStructure(this ILinks<ulong> links, StringBuilder sb,
        ↪ HashSet<ulong> visited, ulong linkIndex, Func<UInt64Link, bool> isElement,
        ↪ Action<StringBuilder, UInt64Link> appendElement, bool renderIndex = false, bool
        ↪ renderDebug = false)
        {
            if (sb == null)
            {
                throw new ArgumentNullException(nameof(sb));
            }
            if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
            ↪ Constants.Itself)
            {
                return;
            }
            if (links.Exists(linkIndex))
            {
                if (visited.Add(linkIndex))
                {
                    sb.Append('(');
                    var link = new UInt64Link(links.GetLink(linkIndex));
                    if (renderIndex)
                    {
                        sb.Append(link.Index);
                        sb.Append(':');
                    }
                    if (link.Source == link.Index)
```

```
                    {
                        sb.Append(link.Index);
                    }
                    else
                    {
                        var source = new UInt64Link(links.GetLink(link.Source));
                        if (isElement(source))
                        {
                            appendElement(sb, source);
                        }
                        else
                        {
                            links.AppendStructure(sb, visited, source.Index, isElement,
                            ↪  appendElement, renderIndex);
                        }
                    }
                    sb.Append(' ');
                    if (link.Target == link.Index)
                    {
                        sb.Append(link.Index);
                    }
                    else
                    {
                        var target = new UInt64Link(links.GetLink(link.Target));
                        if (isElement(target))
                        {
                            appendElement(sb, target);
                        }
                        else
                        {
                            links.AppendStructure(sb, visited, target.Index, isElement,
                            ↪  appendElement, renderIndex);
                        }
                    }
                    sb.Append(')');
                }
                else
                {
                    if (renderDebug)
                    {
                        sb.Append('*');
                    }
                    sb.Append(linkIndex);
                }
            }
            else
            {
                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,
        ///             // для значения в 2 бита, которое представляет тип операции
        ///             throw new NotImplementedException();
        ///         }
        ///     }
        /// }
        ///
        /// private struct Transition
        /// {
        ///     public TransitionHeader Header;
        ///     public Link Source;
        ///     public Link Linker;
        ///     public Link Target;
        /// }
        ///
        /// </remarks>
        public struct Transition
        {
            public static readonly long Size = Structure<Transition>.Size;

            public readonly ulong TransactionId;
            public readonly UInt64Link Before;
            public readonly UInt64Link After;
            public readonly Timestamp Timestamp;

            public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
             ↪  transactionId, UInt64Link before, UInt64Link after)
            {
                TransactionId = transactionId;
                Before = before;
                After = after;
                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();
```

```
                    _layer._transitions.Enqueue(transition);
                }
                _layer._lastCommitedTransactionId = _layer._currentTransactionId;
                IsCommitted = true;
            }

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

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

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

            protected override void Dispose(bool manual, bool wasDisposed)
            {
                if (!wasDisposed && _layer != null && !_layer.IsDisposed)
                {
                    if (!IsCommitted && !IsReverted)
                    {
                        Revert();
                    }
                    _layer.ResetCurrentTransation();
                }
            }
        }

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

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

        public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
            : base(links)
        {
            if (string.IsNullOrWhiteSpace(logAddress))
            {
                throw new ArgumentNullException(nameof(logAddress));
            }
            // В первой строке файла хранится последняя закоммиченную транзакцию.
            // При запуске это используется для проверки удачного закрытия файла лога.
            // In the first line of the file the last committed transaction is stored.
            // On startup, this is used to check that the log file is successfully closed.
            var lastCommitedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
            var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
            if (!lastCommitedTransition.Equals(lastWrittenTransition))
            {
```

```
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));
291         }
292 
293         [MethodImpl(MethodImplOptions.AggressiveInlining)]
294         private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
            ↪  _transitions;
295 
296         private void CommitTransition(Transition transition)
297         {
298             if (_currentTransaction != null)
299             {
300                 Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
301             }
302             var transitions = GetCurrentTransitions();
303             transitions.Enqueue(transition);
304         }
305 
306         private void RevertTransition(Transition transition)
307         {
308             if (transition.After.IsNull()) // Revert Deletion with Creation
309             {
310                 Links.Create();
311             }
312             else if (transition.Before.IsNull()) // Revert Creation with Deletion
313             {
314                 Links.Delete(transition.After.Index);
315             }
316             else // Revert Update
317             {
318                 Links.Update(new[] { transition.After.Index, transition.Before.Source,
                    ↪  transition.Before.Target });
```

```
            }
        }

        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/UnaryNumbers/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
```

```
7
8  namespace Platform.Data.Doublets.UnaryNumbers
9  {
10     public class AddressToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
       ↪  IConverter<TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
13
14         private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;
15
16         public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
           ↪  powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
           ↪  powerOf2ToUnaryNumberConverter;
17
18         public TLink Convert(TLink sourceAddress)
19         {
20             var number = sourceAddress;
21             var nullConstant = Links.Constants.Null;
22             var one = Integer<TLink>.One;
23             var target = nullConstant;
24             for (int i = 0; !_equalityComparer.Equals(number, default) && i <
               ↪  Type<TLink>.BitsLength; i++)
25             {
26                 if (_equalityComparer.Equals(Arithmetic.And(number, one), one))
27                 {
28                     target = _equalityComparer.Equals(target, nullConstant)
29                         ? _powerOf2ToUnaryNumberConverter.Convert(i)
30                         : Links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
31                 }
32                 number = (Integer<TLink>)((ulong)(Integer<TLink>)number >> 1); // Should be
                   ↪  Bit.ShiftRight(number, 1)
33             }
34             return target;
35         }
36     }
37 }
```

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

```
1  using System;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.UnaryNumbers
8  {
9      public class LinkToItsFrequencyNumberConveter<TLink> : LinksOperatorBase<TLink>,
       ↪  IConverter<Doublet<TLink>, TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
12
13         private readonly IPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
14         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
15
16         public LinkToItsFrequencyNumberConveter(
17             ILinks<TLink> links,
18             IPropertyOperator<TLink, TLink> frequencyPropertyOperator,
19             IConverter<TLink> unaryNumberToAddressConverter)
20             : base(links)
21         {
22             _frequencyPropertyOperator = frequencyPropertyOperator;
23             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
24         }
25
26         public TLink Convert(Doublet<TLink> doublet)
27         {
28             var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
29             if (_equalityComparer.Equals(link, default))
30             {
31                 throw new ArgumentException($"Link ({doublet}) not found.", nameof(doublet));
32             }
33             var frequency = _frequencyPropertyOperator.Get(link);
34             if (_equalityComparer.Equals(frequency, default))
35             {
36                 return default;
37             }
38             var frequencyNumber = Links.GetSource(frequency);
39             return _unaryNumberToAddressConverter.Convert(frequencyNumber);
```

```
        }
    }
}
```

./Platform.Data.Doublets/UnaryNumbers/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.UnaryNumbers
{
    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/UnaryNumbers/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.UnaryNumbers
{
    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/UnaryNumbers/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.UnaryNumbers
{
    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/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> _addressToUnaryNumberConverter;
        private readonly TLink _unicodeSymbolMarker;

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

        public TLink Convert(char source)
        {
            var unaryNumber = _addressToUnaryNumberConverter.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)

```

```
62          [MethodImpl(MethodImplOptions.AggressiveInlining)]
63          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;
```

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

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

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

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

#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> _unaryNumberToAddressConverter;
        private readonly ICriterionMatcher<TLink> _unicodeSymbolCriterionMatcher;

        public UnicodeSymbolToCharConverter(ILinks<TLink> links, IConverter<TLink> unaryNumberToAddressConverter, ICriterionMatcher<TLink> unicodeSymbolCriterionMatcher) : base(links)
        {
            _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
```

```
            _unicodeSymbolCriterionMatcher = unicodeSymbolCriterionMatcher;
        }

        public char Convert(TLink source)
        {
            if (!_unicodeSymbolCriterionMatcher.IsMatched(source))
            {
                throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
                ↪  not a unicode symbol.");
            }
            return (char)(ushort)(Integer<TLink>)_unaryNumberToAddressConverter.Convert(Links.Ge
            ↪  tSource(source));
        }
    }
}
```

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

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

namespace Platform.Data.Doublets.Tests
{
    public static class ComparisonTests
    {
        protected class UInt64Comparer : IComparer<ulong>
        {
            public int Compare(ulong x, ulong y) => x.CompareTo(y);
        }

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

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

            ulong x = 10;
            ulong y = 500;

            bool result = false;

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

            var comparer1 = Comparer<ulong>.Default;

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

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

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

            var comparer2 = new UInt64Comparer();

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

```
64
65              Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {result}");
66          }
67      }
68  }
```

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

```
1   using System.Collections.Generic;
2   using Xunit;
3   using Platform.Reflection;
4   using Platform.Numbers;
5   using Platform.Memory;
6   using Platform.Scopes;
7   using Platform.Setters;
8   using Platform.Data.Doublets.ResizableDirectMemory;
9
10  namespace Platform.Data.Doublets.Tests
11  {
12      public static class DoubletLinksTests
13      {
14          [Fact]
15          public static void UInt64CRUDTest()
16          {
17              using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪  ResizableDirectMemoryLinks<ulong>>>())
18              {
19                  scope.Use<ILinks<ulong>>().TestCRUDOperations();
20              }
21          }
22
23          [Fact]
24          public static void UInt32CRUDTest()
25          {
26              using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪  ResizableDirectMemoryLinks<uint>>>())
27              {
28                  scope.Use<ILinks<uint>>().TestCRUDOperations();
29              }
30          }
31
32          [Fact]
33          public static void UInt16CRUDTest()
34          {
35              using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪  ResizableDirectMemoryLinks<ushort>>>())
36              {
37                  scope.Use<ILinks<ushort>>().TestCRUDOperations();
38              }
39          }
40
41          [Fact]
42          public static void UInt8CRUDTest()
43          {
44              using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪  ResizableDirectMemoryLinks<byte>>>())
45              {
46                  scope.Use<ILinks<byte>>().TestCRUDOperations();
47              }
48          }
49
50          private static void TestCRUDOperations<T>(this ILinks<T> links)
51          {
52              var constants = links.Constants;
53
54              var equalityComparer = EqualityComparer<T>.Default;
55
56              // Create Link
57              Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));
58
59              var setter = new Setter<T>(constants.Null);
60              links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
61
62              Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
63
64              var linkAddress = links.Create();
65
66              var link = new Link<T>(links.GetLink(linkAddress));
67
68              Assert.True(link.Count == 3);
```

```
69              Assert.True(equalityComparer.Equals(link.Index, linkAddress));
70              Assert.True(equalityComparer.Equals(link.Source, constants.Null));
71              Assert.True(equalityComparer.Equals(link.Target, constants.Null));
72
73              Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.One));
74
75              // Get first link
76              setter = new Setter<T>(constants.Null);
77              links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);
78
79              Assert.True(equalityComparer.Equals(setter.Result, linkAddress));
80
81              // Update link to reference itself
82              links.Update(linkAddress, linkAddress, linkAddress);
83
84              link = new Link<T>(links.GetLink(linkAddress));
85
86              Assert.True(equalityComparer.Equals(link.Source, linkAddress));
87              Assert.True(equalityComparer.Equals(link.Target, linkAddress));
88
89              // Update link to reference null (prepare for delete)
90              var updated = links.Update(linkAddress, constants.Null, constants.Null);
91
92              Assert.True(equalityComparer.Equals(updated, linkAddress));
93
94              link = new Link<T>(links.GetLink(linkAddress));
95
96              Assert.True(equalityComparer.Equals(link.Source, constants.Null));
97              Assert.True(equalityComparer.Equals(link.Target, constants.Null));
98
99              // Delete link
100             links.Delete(linkAddress);
101
102             Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));
103
104             setter = new Setter<T>(constants.Null);
105             links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
106
107             Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
108         }
109
110         [Fact]
111         public static void UInt64RawNumbersCRUDTest()
112         {
113             using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪ ResizableDirectMemoryLinks<ulong>>>())
114             {
115                 scope.Use<ILinks<ulong>>().TestRawNumbersCRUDOperations();
116             }
117         }
118
119         [Fact]
120         public static void UInt32RawNumbersCRUDTest()
121         {
122             using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪ ResizableDirectMemoryLinks<uint>>>())
123             {
124                 scope.Use<ILinks<uint>>().TestRawNumbersCRUDOperations();
125             }
126         }
127
128         [Fact]
129         public static void UInt16RawNumbersCRUDTest()
130         {
131             using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪ ResizableDirectMemoryLinks<ushort>>>())
132             {
133                 scope.Use<ILinks<ushort>>().TestRawNumbersCRUDOperations();
134             }
135         }
136
137         [Fact]
138         public static void UInt8RawNumbersCRUDTest()
139         {
140             using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪ ResizableDirectMemoryLinks<byte>>>())
141             {
142                 scope.Use<ILinks<byte>>().TestRawNumbersCRUDOperations();
143             }
```

```
144         }
145
146         private static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
147         {
148             // Constants
149             var constants = links.Constants;
150             var equalityComparer = EqualityComparer<T>.Default;
151
152             var h106E = new Hybrid<T>(106L, isExternal: true);
153             var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
154             var h108E = new Hybrid<T>(-108L);
155
156             Assert.Equal(106L, h106E.AbsoluteValue);
157             Assert.Equal(107L, h107E.AbsoluteValue);
158             Assert.Equal(108L, h108E.AbsoluteValue);
159
160             // Create Link (External -> External)
161             var linkAddress1 = links.Create();
162
163             links.Update(linkAddress1, h106E, h108E);
164
165             var link1 = new Link<T>(links.GetLink(linkAddress1));
166
167             Assert.True(equalityComparer.Equals(link1.Source, h106E));
168             Assert.True(equalityComparer.Equals(link1.Target, h108E));
169
170             // Create Link (Internal -> External)
171             var linkAddress2 = links.Create();
172
173             links.Update(linkAddress2, linkAddress1, h108E);
174
175             var link2 = new Link<T>(links.GetLink(linkAddress2));
176
177             Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
178             Assert.True(equalityComparer.Equals(link2.Target, h108E));
179
180             // Create Link (Internal -> Internal)
181             var linkAddress3 = links.Create();
182
183             links.Update(linkAddress3, linkAddress1, linkAddress2);
184
185             var link3 = new Link<T>(links.GetLink(linkAddress3));
186
187             Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
188             Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));
189
190             // Search for created link
191             var setter1 = new Setter<T>(constants.Null);
192             links.Each(h106E, h108E, setter1.SetAndReturnFalse);
193
194             Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));
195
196             // Search for nonexistent link
197             var setter2 = new Setter<T>(constants.Null);
198             links.Each(h106E, h107E, setter2.SetAndReturnFalse);
199
200             Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));
201
202             // Update link to reference null (prepare for delete)
203             var updated = links.Update(linkAddress3, constants.Null, constants.Null);
204
205             Assert.True(equalityComparer.Equals(updated, linkAddress3));
206
207             link3 = new Link<T>(links.GetLink(linkAddress3));
208
209             Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
210             Assert.True(equalityComparer.Equals(link3.Target, constants.Null));
211
212             // Delete link
213             links.Delete(linkAddress3);
214
215             Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Two));
216
217             var setter3 = new Setter<T>(constants.Null);
218             links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);
219
220             Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
221         }
222
223         // TODO: Test layers
```

```
    }
}
```

./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);
                }
            });

```

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

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

            var comparer = Comparer<ulong>.Default;

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

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

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

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

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Xunit;
using Platform.Disposables;
using Platform.IO;
using Platform.Ranges;
using Platform.Random;
using Platform.Timestamps;
using Platform.Singletons;
using Platform.Counters;
using Platform.Diagnostics;
using Platform.Data.Constants;
using Platform.Data.Doublets.ResizableDirectMemory;
using Platform.Data.Doublets.Decorators;

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

        private const long Iterations = 10 * 1024;

        #region Concept

        [Fact]
        public static void MultipleCreateAndDeleteTest()
        {
            //const int N = 21;

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

                for (var N = 0; N < 100; N++)
                {
                    var random = new System.Random(N);

                    var created = 0;
                    var deleted = 0;

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

```
50
51                      var createPoint = random.NextBoolean();
52
53                      if (linksCount > 2 && createPoint)
54                      {
55                          var linksAddressRange = new Range<ulong>(1, linksCount);
56                          var source = random.NextUInt64(linksAddressRange);
57                          var target = random.NextUInt64(linksAddressRange); //-V3086
58
59                          var resultLink = links.CreateAndUpdate(source, target);
60                          if (resultLink > linksCount)
61                          {
62                              created++;
63                          }
64                      }
65                      else
66                      {
67                          links.Create();
68                          created++;
69                      }
70                  }
71
72                  Assert.True(created == (int)links.Count());
73
74                  for (var i = 0; i < N; i++)
75                  {
76                      var link = (ulong)i + 1;
77                      if (links.Exists(link))
78                      {
79                          links.Delete(link);
80                          deleted++;
81                      }
82                  }
83
84                  Assert.True(links.Count() == 0);
85              }
86          }
87      }
88
89      [Fact]
90      public static void CascadeUpdateTest()
91      {
92          var itself = _constants.Itself;
93
94          using (var scope = new TempLinksTestScope(useLog: true))
95          {
96              var links = scope.Links;
97
98              var l1 = links.Create();
99              var l2 = links.Create();
100
101             l2 = links.Update(l2, l2, l1, l2);
102
103             links.CreateAndUpdate(l2, itself);
104             links.CreateAndUpdate(l2, itself);
105
106             l2 = links.Update(l2, l1);
107
108             links.Delete(l2);
109
110             Global.Trash = links.Count();
111
112             links.Unsync.DisposeIfPossible(); // Close links to access log
113
114             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop
                ↪ e.TempTransactionLogFilename);
115         }
116     }
117
118     [Fact]
119     public static void BasicTransactionLogTest()
120     {
121         using (var scope = new TempLinksTestScope(useLog: true))
122         {
123             var links = scope.Links;
124             var l1 = links.Create();
125             var l2 = links.Create();
126
127             Global.Trash = links.Update(l2, l2, l1, l2);
128
```

```
129                 links.Delete(l1);
130
131                 links.Unsync.DisposeIfPossible(); // Close links to access log
132
133                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop
                    ↪  e.TempTransactionLogFilename);
134             }
135         }
136
137         [Fact]
138         public static void TransactionAutoRevertedTest()
139         {
140             // Auto Reverted (Because no commit at transaction)
141             using (var scope = new TempLinksTestScope(useLog: true))
142             {
143                 var links = scope.Links;
144                 var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
145                 using (var transaction = transactionsLayer.BeginTransaction())
146                 {
147                     var l1 = links.Create();
148                     var l2 = links.Create();
149
150                     links.Update(l2, l2, l1, l2);
151                 }
152
153                 Assert.Equal(0UL, links.Count());
154
155                 links.Unsync.DisposeIfPossible();
156
157                 var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s
                    ↪  cope.TempTransactionLogFilename);
158                 Assert.Single(transitions);
159             }
160         }
161
162         [Fact]
163         public static void TransactionUserCodeErrorNoDataSavedTest()
164         {
165             // User Code Error (Autoreverted), no data saved
166             var itself = _constants.Itself;
167
168             TempLinksTestScope lastScope = null;
169             try
170             {
171                 using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
                    ↪  useLog: true))
172                 {
173                     var links = scope.Links;
174                     var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor
                        ↪  atorBase<ulong>)links.Unsync).Links;
175                     using (var transaction = transactionsLayer.BeginTransaction())
176                     {
177                         var l1 = links.CreateAndUpdate(itself, itself);
178                         var l2 = links.CreateAndUpdate(itself, itself);
179
180                         l2 = links.Update(l2, l2, l1, l2);
181
182                         links.CreateAndUpdate(l2, itself);
183                         links.CreateAndUpdate(l2, itself);
184
185                         //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi
                            ↪  tion>(scope.TempTransactionLogFilename);
186
187                         l2 = links.Update(l2, l1);
188
189                         links.Delete(l2);
190
191                         ExceptionThrower();
192
193                         transaction.Commit();
194                     }
195
196                     Global.Trash = links.Count();
197                 }
198             }
199             catch
200             {
201                 Assert.False(lastScope == null);
202
```

```
203                 var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l
                    ↪  astScope.TempTransactionLogFilename);
204
205                 Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
                    ↪  transitions[0].After.IsNull());
206
207                 lastScope.DeleteFiles();
208             }
209         }
210
211         [Fact]
212         public static void TransactionUserCodeErrorSomeDataSavedTest()
213         {
214             // User Code Error (Autoreverted), some data saved
215             var itself = _constants.Itself;
216
217             TempLinksTestScope lastScope = null;
218             try
219             {
220                 ulong l1;
221                 ulong l2;
222
223                 using (var scope = new TempLinksTestScope(useLog: true))
224                 {
225                     var links = scope.Links;
226                     l1 = links.CreateAndUpdate(itself, itself);
227                     l2 = links.CreateAndUpdate(itself, itself);
228
229                     l2 = links.Update(l2, l2, l1, l2);
230
231                     links.CreateAndUpdate(l2, itself);
232                     links.CreateAndUpdate(l2, itself);
233
234                     links.Unsync.DisposeIfPossible();
235
236                     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
                        ↪  scope.TempTransactionLogFilename);
237                 }
238
239                 using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
                    ↪  useLog: true))
240                 {
241                     var links = scope.Links;
242                     var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
243                     using (var transaction = transactionsLayer.BeginTransaction())
244                     {
245                         l2 = links.Update(l2, l1);
246
247                         links.Delete(l2);
248
249                         ExceptionThrower();
250
251                         transaction.Commit();
252                     }
253
254                     Global.Trash = links.Count();
255                 }
256             }
257             catch
258             {
259                 Assert.False(lastScope == null);
260
261                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
                    ↪  Scope.TempTransactionLogFilename);
262
263                 lastScope.DeleteFiles();
264             }
265         }
266
267         [Fact]
268         public static void TransactionCommit()
269         {
270             var itself = _constants.Itself;
271
272             var tempDatabaseFilename = Path.GetTempFileName();
273             var tempTransactionLogFilename = Path.GetTempFileName();
274
275             // Commit
```

```
276             using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                ↪ tempTransactionLogFilename))
277             using (var links = new UInt64Links(memoryAdapter))
278             {
279                 using (var transaction = memoryAdapter.BeginTransaction())
280                 {
281                     var l1 = links.CreateAndUpdate(itself, itself);
282                     var l2 = links.CreateAndUpdate(itself, itself);
283 
284                     Global.Trash = links.Update(l2, l2, l1, l2);
285 
286                     links.Delete(l1);
287 
288                     transaction.Commit();
289                 }
290 
291                 Global.Trash = links.Count();
292             }
293 
294             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                ↪ sactionLogFilename);
295         }
296 
297         [Fact]
298         public static void TransactionDamage()
299         {
300             var itself = _constants.Itself;
301 
302             var tempDatabaseFilename = Path.GetTempFileName();
303             var tempTransactionLogFilename = Path.GetTempFileName();
304 
305             // Commit
306             using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                ↪ tempTransactionLogFilename))
307             using (var links = new UInt64Links(memoryAdapter))
308             {
309                 using (var transaction = memoryAdapter.BeginTransaction())
310                 {
311                     var l1 = links.CreateAndUpdate(itself, itself);
312                     var l2 = links.CreateAndUpdate(itself, itself);
313 
314                     Global.Trash = links.Update(l2, l2, l1, l2);
315 
316                     links.Delete(l1);
317 
318                     transaction.Commit();
319                 }
320 
321                 Global.Trash = links.Count();
322             }
323 
324             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                ↪ sactionLogFilename);
325 
326             // Damage database
327 
328             FileHelpers.WriteFirst(tempTransactionLogFilename, new
                ↪ UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
329 
330             // Try load damaged database
331             try
332             {
333                 // TODO: Fix
334                 using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                    ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                    ↪ tempTransactionLogFilename))
335                 using (var links = new UInt64Links(memoryAdapter))
336                 {
337                     Global.Trash = links.Count();
338                 }
339             }
340             catch (NotSupportedException ex)
341             {
342                 Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
                    ↪ yet.");
343             }
```

```
344
345             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                ↪  sactionLogFilename);
346
347             File.Delete(tempDatabaseFilename);
348             File.Delete(tempTransactionLogFilename);
349         }
350
351         [Fact]
352         public static void Bug1Test()
353         {
354             var tempDatabaseFilename = Path.GetTempFileName();
355             var tempTransactionLogFilename = Path.GetTempFileName();
356
357             var itself = _constants.Itself;
358
359             // User Code Error (Autoreverted), some data saved
360             try
361             {
362                 ulong l1;
363                 ulong l2;
364
365                 using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                    ↪  UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                    ↪  tempTransactionLogFilename))
366                 using (var links = new UInt64Links(memoryAdapter))
367                 {
368                     l1 = links.CreateAndUpdate(itself, itself);
369                     l2 = links.CreateAndUpdate(itself, itself);
370
371                     l2 = links.Update(l2, l2, l1, l2);
372
373                     links.CreateAndUpdate(l2, itself);
374                     links.CreateAndUpdate(l2, itself);
375                 }
376
377                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
                    ↪  TransactionLogFilename);
378
379                 using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                    ↪  UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                    ↪  tempTransactionLogFilename))
380                 using (var links = new UInt64Links(memoryAdapter))
381                 {
382                     using (var transaction = memoryAdapter.BeginTransaction())
383                     {
384                         l2 = links.Update(l2, l1);
385
386                         links.Delete(l2);
387
388                         ExceptionThrower();
389
390                         transaction.Commit();
391                     }
392
393                     Global.Trash = links.Count();
394                 }
395             }
396             catch
397             {
398                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
                    ↪  TransactionLogFilename);
399             }
400
401             File.Delete(tempDatabaseFilename);
402             File.Delete(tempTransactionLogFilename);
403         }
404
405         private static void ExceptionThrower()
406         {
407             throw new Exception();
408         }
409
410         [Fact]
411         public static void PathsTest()
412         {
413             var source = _constants.SourcePart;
414             var target = _constants.TargetPart;
415
```

```
416             using (var scope = new TempLinksTestScope())
417             {
418                 var links = scope.Links;
419                 var l1 = links.CreatePoint();
420                 var l2 = links.CreatePoint();
421 
422                 var r1 = links.GetByKeys(l1, source, target, source);
423                 var r2 = links.CheckPathExistance(l2, l2, l2, l2);
424             }
425         }
426 
427         [Fact]
428         public static void RecursiveStringFormattingTest()
429         {
430             using (var scope = new TempLinksTestScope(useSequences: true))
431             {
432                 var links = scope.Links;
433                 var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.
434 
435                 var a = links.CreatePoint();
436                 var b = links.CreatePoint();
437                 var c = links.CreatePoint();
438 
439                 var ab = links.CreateAndUpdate(a, b);
440                 var cb = links.CreateAndUpdate(c, b);
441                 var ac = links.CreateAndUpdate(a, c);
442 
443                 a = links.Update(a, c, b);
444                 b = links.Update(b, a, c);
445                 c = links.Update(c, a, b);
446 
447                 Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
448                 Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
449                 Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
450 
451                 Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
                    ↪  "(5:(4:5 (6:5 4)) 6)");
452                 Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
                    ↪  "(6:(5:(4:5 6) 6) 4)");
453                 Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
                    ↪  "(4:(5:4 (6:5 4)) 6)");
454 
455                 // TODO: Think how to build balanced syntax tree while formatting structure (eg.
                    ↪  "(4:(5:4 6) (6:5 4)") instead of "(4:(5:4 (6:5 4)) 6)"
456 
457                 Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
                    ↪  "{{5}{5}{4}{6}}");
458                 Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
                    ↪  "{{5}{6}{6}{4}}");
459                 Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
                    ↪  "{{4}{5}{4}{6}}");
460             }
461         }
462 
463         private static void DefaultFormatter(StringBuilder sb, ulong link)
464         {
465             sb.Append(link.ToString());
466         }
467 
468         #endregion
469 
470         #region Performance
471 
472         /*
473         public static void RunAllPerformanceTests()
474         {
475             try
476             {
477                 links.TestLinksInSteps();
478             }
479             catch (Exception ex)
480             {
481                 ex.WriteToConsole();
482             }
483 
484             return;
485 
486             try
487             {
```

```
488                 //ThreadPool.SetMaxThreads(2, 2);
489
490                 // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
    ↪  результат
491                 // Также это дополнительно помогает в отладке
492                 // Увеличивает вероятность попадания информации в кэши
493                 for (var i = 0; i < 10; i++)
494                 {
495                     //0 - 10 ГБ
496                     //Каждые 100 МБ срез цифр
497
498                     //links.TestGetSourceFunction();
499                     //links.TestGetSourceFunctionInParallel();
500                     //links.TestGetTargetFunction();
501                     //links.TestGetTargetFunctionInParallel();
502                     links.Create64BillionLinks();
503
504                     links.TestRandomSearchFixed();
505                     //links.Create64BillionLinksInParallel();
506                     links.TestEachFunction();
507                     //links.TestForeach();
508                     //links.TestParallelForeach();
509                 }
510
511                 links.TestDeletionOfAllLinks();
512
513             }
514             catch (Exception ex)
515             {
516                 ex.WriteToConsole();
517             }
518         }*/
519
520          /*
521         public static void TestLinksInSteps()
522         {
523             const long gibibyte = 1024 * 1024 * 1024;
524             const long mebibyte = 1024 * 1024;
525
526             var totalLinksToCreate = gibibyte /
    ↪  Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
527             var linksStep = 102 * mebibyte /
    ↪  Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
528
529             var creationMeasurements = new List<TimeSpan>();
530             var searchMeasuremets = new List<TimeSpan>();
531             var deletionMeasurements = new List<TimeSpan>();
532
533             GetBaseRandomLoopOverhead(linksStep);
534             GetBaseRandomLoopOverhead(linksStep);
535
536             var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
537
538             ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
539
540             var loops = totalLinksToCreate / linksStep;
541
542             for (int i = 0; i < loops; i++)
543             {
544                 creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
545                 searchMeasuremets.Add(Measure(() => links.RunRandomSearches(linksStep)));
546
547                 Console.Write("\rC + S {0}/{1}", i + 1, loops);
548             }
549
550             ConsoleHelpers.Debug();
551
552             for (int i = 0; i < loops; i++)
553             {
554                 deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
555
556                 Console.Write("\rD {0}/{1}", i + 1, loops);
557             }
558
559             ConsoleHelpers.Debug();
560
561             ConsoleHelpers.Debug("C S D");
562
563             for (int i = 0; i < loops; i++)
564             {
```

```
565                 ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
    ↪  searchMeasuremets[i], deletionMeasurements[i]);
566             }
567
568             ConsoleHelpers.Debug("C S D (no overhead)");
569
570             for (int i = 0; i < loops; i++)
571             {
572                 ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] - stepLoopOverhead,
    ↪  searchMeasuremets[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
573             }
574
575             ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
    ↪  links.Total);
576         }
577
578         private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
    ↪  amountToCreate)
579         {
580             for (long i = 0; i < amountToCreate; i++)
581                 links.Create(0, 0);
582         }
583
584          private static TimeSpan GetBaseRandomLoopOverhead(long loops)
585          {
586              return Measure(() =>
587              {
588                  ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
589                  ulong result = 0;
590                  for (long i = 0; i < loops; i++)
591                  {
592                      var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
593                      var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
594
595                      result += maxValue + source + target;
596                  }
597                  Global.Trash = result;
598              });
599          }
600           */
601
602          [Fact(Skip = "performance test")]
603          public static void GetSourceTest()
604          {
605              using (var scope = new TempLinksTestScope())
606              {
607                  var links = scope.Links;
608                  ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
                     ↪  Iterations);
609
610                  ulong counter = 0;
611
612                  //var firstLink = links.First();
613                  // Создаём одну связь, из которой будет производить считывание
614                  var firstLink = links.Create();
615
616                  var sw = Stopwatch.StartNew();
617
618                  // Тестируем саму функцию
619                  for (ulong i = 0; i < Iterations; i++)
620                  {
621                      counter += links.GetSource(firstLink);
622                  }
623
624                  var elapsedTime = sw.Elapsed;
625
626                  var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
627
628                  // Удаляем связь, из которой производилось считывание
629                  links.Delete(firstLink);
630
631                  ConsoleHelpers.Debug(
632                      "{0} Iterations of GetSource function done in {1} ({2} Iterations per
                         ↪  second), counter result: {3}",
633                      Iterations, elapsedTime, (long)iterationsPerSecond, counter);
634              }
635          }
636
637          [Fact(Skip = "performance test")]
```

```
638         public static void GetSourceInParallel()
639         {
640             using (var scope = new TempLinksTestScope())
641             {
642                 var links = scope.Links;
643                 ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
                    ↪  parallel.", Iterations);
644
645                 long counter = 0;
646
647                 //var firstLink = links.First();
648                 var firstLink = links.Create();
649
650                 var sw = Stopwatch.StartNew();
651
652                 // Тестируем саму функцию
653                 Parallel.For(0, Iterations, x =>
654                 {
655                     Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
656                     //Interlocked.Increment(ref counter);
657                 });
658
659                 var elapsedTime = sw.Elapsed;
660
661                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
662
663                 links.Delete(firstLink);
664
665                 ConsoleHelpers.Debug(
666                     "{0} Iterations of GetSource function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
667                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
668             }
669         }
670
671         [Fact(Skip = "performance test")]
672         public static void TestGetTarget()
673         {
674             using (var scope = new TempLinksTestScope())
675             {
676                 var links = scope.Links;
677                 ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
                    ↪  Iterations);
678
679                 ulong counter = 0;
680
681                 //var firstLink = links.First();
682                 var firstLink = links.Create();
683
684                 var sw = Stopwatch.StartNew();
685
686                 for (ulong i = 0; i < Iterations; i++)
687                 {
688                     counter += links.GetTarget(firstLink);
689                 }
690
691                 var elapsedTime = sw.Elapsed;
692
693                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
694
695                 links.Delete(firstLink);
696
697                 ConsoleHelpers.Debug(
698                     "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
699                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
700             }
701         }
702
703         [Fact(Skip = "performance test")]
704         public static void TestGetTargetInParallel()
705         {
706             using (var scope = new TempLinksTestScope())
707             {
708                 var links = scope.Links;
709                 ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
                    ↪  parallel.", Iterations);
710
711                 long counter = 0;
712
```

```
713                 //var firstLink = links.First();
714                 var firstLink = links.Create();
715 
716                 var sw = Stopwatch.StartNew();
717 
718                 Parallel.For(0, Iterations, x =>
719                 {
720                     Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
721                     //Interlocked.Increment(ref counter);
722                 });
723 
724                 var elapsedTime = sw.Elapsed;
725 
726                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
727 
728                 links.Delete(firstLink);
729 
730                 ConsoleHelpers.Debug(
731                     "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
732                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
733             }
734         }
735 
736         // TODO: Заполнить базу данных перед тестом
737         /*
738         [Fact]
739         public void TestRandomSearchFixed()
740         {
741             var tempFilename = Path.GetTempFileName();
742 
743             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪  DefaultLinksSizeStep))
744             {
745                 long iterations = 64 * 1024 * 1024 /
    ↪  Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
746 
747                 ulong counter = 0;
748                 var maxLink = links.Total;
749 
750                 ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);
751 
752                 var sw = Stopwatch.StartNew();
753 
754                 for (var i = iterations; i > 0; i--)
755                 {
756                     var source =
    ↪  RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
757                     var target =
    ↪  RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
758 
759                     counter += links.Search(source, target);
760                 }
761 
762                 var elapsedTime = sw.Elapsed;
763 
764                 var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
765 
766                 ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
    ↪  Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
    ↪  counter);
767             }
768 
769             File.Delete(tempFilename);
770         }*/
771 
772         [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
773         public static void TestRandomSearchAll()
774         {
775             using (var scope = new TempLinksTestScope())
776             {
777                 var links = scope.Links;
778                 ulong counter = 0;
779 
780                 var maxLink = links.Count();
781 
782                 var iterations = links.Count();
783 
784                 ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
                    ↪  links.Count());
```

```
785
786                 var sw = Stopwatch.StartNew();
787
788                 for (var i = iterations; i > 0; i--)
789                 {
790                     var linksAddressRange = new Range<ulong>(_constants.MinPossibleIndex,
                        ↪  maxLink);
791
792                     var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
793                     var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
794
795                     counter += links.SearchOrDefault(source, target);
796                 }
797
798                 var elapsedTime = sw.Elapsed;
799
800                 var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
801
802                 ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
                    ↪  Iterations per second), c: {3}",
803                      iterations, elapsedTime, (long)iterationsPerSecond, counter);
804             }
805         }
806
807         [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
808         public static void TestEach()
809         {
810             using (var scope = new TempLinksTestScope())
811             {
812                 var links = scope.Links;
813
814                 var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
815
816                 ConsoleHelpers.Debug("Testing Each function.");
817
818                 var sw = Stopwatch.StartNew();
819
820                 links.Each(counter.IncrementAndReturnTrue);
821
822                 var elapsedTime = sw.Elapsed;
823
824                 var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
825
826                 ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
                    ↪  links per second)",
827                     counter, elapsedTime, (long)linksPerSecond);
828             }
829         }
830
831         /*
832         [Fact]
833         public static void TestForeach()
834         {
835             var tempFilename = Path.GetTempFileName();
836
837             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪  DefaultLinksSizeStep))
838             {
839                 ulong counter = 0;
840
841                 ConsoleHelpers.Debug("Testing foreach through links.");
842
843                 var sw = Stopwatch.StartNew();
844
845                 //foreach (var link in links)
846                 //{
847                 //    counter++;
848                 //}
849
850                 var elapsedTime = sw.Elapsed;
851
852                 var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
853
854                 ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
    ↪  links per second)", counter, elapsedTime, (long)linksPerSecond);
855             }
856
857             File.Delete(tempFilename);
858         }
859         */
```

```
860
861         /*
862         [Fact]
863         public static void TestParallelForeach()
864         {
865             var tempFilename = Path.GetTempFileName();
866
867             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪  DefaultLinksSizeStep))
868             {
869
870                 long counter = 0;
871
872                 ConsoleHelpers.Debug("Testing parallel foreach through links.");
873
874                 var sw = Stopwatch.StartNew();
875
876                 //Parallel.ForEach((IEnumerable<ulong>)links, x =>
877                 //{
878                 //    Interlocked.Increment(ref counter);
879                 //});
880
881                 var elapsedTime = sw.Elapsed;
882
883                 var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
884
885                 ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
    ↪  {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
886             }
887
888             File.Delete(tempFilename);
889         }
890         */
891
892         [Fact(Skip = "performance test")]
893         public static void Create64BillionLinks()
894         {
895             using (var scope = new TempLinksTestScope())
896             {
897                 var links = scope.Links;
898                 var linksBeforeTest = links.Count();
899
900                 long linksToCreate = 64 * 1024 * 1024 /
                    ↪  UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
901
902                 ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
903
904                 var elapsedTime = Performance.Measure(() =>
905                 {
906                     for (long i = 0; i < linksToCreate; i++)
907                     {
908                         links.Create();
909                     }
910                 });
911
912                 var linksCreated = links.Count() - linksBeforeTest;
913                 var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
914
915                 ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
916
917                 ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
                    ↪  linksCreated, elapsedTime,
918                     (long)linksPerSecond);
919             }
920         }
921
922         [Fact(Skip = "performance test")]
923         public static void Create64BillionLinksInParallel()
924         {
925             using (var scope = new TempLinksTestScope())
926             {
927                 var links = scope.Links;
928                 var linksBeforeTest = links.Count();
929
930                 var sw = Stopwatch.StartNew();
931
932                 long linksToCreate = 64 * 1024 * 1024 /
                    ↪  UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
933
```

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

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

                var elapsedTime = sw.Elapsed;

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

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

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

                ConsoleHelpers.Debug("Deleting all links");

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

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

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

        #endregion
    }
}
```

./Platform.Data.Doublets.Tests/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.UnaryNumbers;

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);

```

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

```
}
```

./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);


    //        SequencesOptions o = new SequencesOptions();

    // TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
    //        o.


    //        var sequences = new Sequences(links);

    //        var sw1 = Stopwatch.StartNew();
    //        var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();

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

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

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

    //    File.Delete(tempFilename);
    //}

    [Fact]
    public static void AllVariantsSearchTest()
    {
        const long sequenceLength = 8;

        using (var scope = new TempLinksTestScope(useSequences: true))
        {
            var links = scope.Links;
            var sequences = scope.Sequences;

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

            var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();

            //for (int i = 0; i < createResults.Length; i++)
            //    sequences.Create(createResults[i]);

            var sw0 = Stopwatch.StartNew();
            var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();

            var sw1 = Stopwatch.StartNew();
```

```
            var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();

            var sw2 = Stopwatch.StartNew();
            var searchResults2 = sequences.Each1(sequence); sw2.Stop();

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

            var intersection0 = createResults.Intersect(searchResults0).ToList();
            Assert.True(intersection0.Count == searchResults0.Count);
            Assert.True(intersection0.Count == createResults.Length);

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

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

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

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

    [Fact]
    public static void BalancedVariantSearchTest()
    {
        const long sequenceLength = 200;

        using (var scope = new TempLinksTestScope(useSequences: true))
        {
            var links = scope.Links;
            var sequences = scope.Sequences;

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

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

            var sw1 = Stopwatch.StartNew();
            var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();

            var sw2 = Stopwatch.StartNew();
            var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();

            var sw3 = Stopwatch.StartNew();
            var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();

            // На количестве в 200 элементов это будет занимать вечность
            //var sw4 = Stopwatch.StartNew();
            //var searchResults4 = sequences.Each(sequence); sw4.Stop();

            Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);

            Assert.True(searchResults3.Count == 1 && balancedVariant ==
            ↪  searchResults3.First());

            //Assert.True(sw1.Elapsed < sw2.Elapsed);

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

    [Fact]
    public static void AllPartialVariantsSearchTest()
    {
        const long sequenceLength = 8;
```

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

```
282                 var partialSequence = new ulong[sequenceLength - 2];
283
284                 Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
285
286                 var sw1 = Stopwatch.StartNew();
287                 var searchResults1 =
                    ↪  sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
288
289                 var sw2 = Stopwatch.StartNew();
290                 var searchResults2 =
                    ↪  sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
291
292                 Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
293
294                 Assert.True(searchResults2.Count == 1 && balancedVariant ==
                    ↪  searchResults2.First());
295
296                 for (var i = 0; i < sequenceLength; i++)
297                 {
298                     links.Delete(sequence[i]);
299                 }
300             }
301         }
302
303         [Fact(Skip = "Correct implementation is pending")]
304         public static void PatternMatchTest()
305         {
306             var zeroOrMany = Sequences.Sequences.ZeroOrMany;
307
308             using (var scope = new TempLinksTestScope(useSequences: true))
309             {
310                 var links = scope.Links;
311                 var sequences = scope.Sequences;
312
313                 var e1 = links.Create();
314                 var e2 = links.Create();
315
316                 var sequence = new[]
317                 {
318                     e1, e2, e1, e2 // mama / papa
319                 };
320
321                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
322
323                 var balancedVariant = balancedVariantConverter.Convert(sequence);
324
325                 // 1: [1]
326                 // 2: [2]
327                 // 3: [1,2]
328                 // 4: [1,2,1,2]
329
330                 var doublet = links.GetSource(balancedVariant);
331
332                 var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);
333
334                 Assert.True(matchedSequences1.Count == 0);
335
336                 var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);
337
338                 Assert.True(matchedSequences2.Count == 0);
339
340                 var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);
341
342                 Assert.True(matchedSequences3.Count == 0);
343
344                 var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
345
346                 Assert.Contains(doublet, matchedSequences4);
347                 Assert.Contains(balancedVariant, matchedSequences4);
348
349                 for (var i = 0; i < sequence.Length; i++)
350                 {
351                     links.Delete(sequence[i]);
352                 }
353             }
354         }
355
356         [Fact]
357         public static void IndexTest()
358         {
```

```
359             using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
                ↪ true }, useSequences: true))
360             {
361                 var links = scope.Links;
362                 var sequences = scope.Sequences;
363                 var index = sequences.Options.Index;
364
365                 var e1 = links.Create();
366                 var e2 = links.Create();
367
368                 var sequence = new[]
369                 {
370                     e1, e2, e1, e2 // mama / papa
371                 };
372
373                 Assert.False(index.MightContain(sequence));
374
375                 index.Add(sequence);
376
377                 Assert.True(index.MightContain(sequence));
378             }
379         }
380
381         /// <summary>Imported from https://raw.githubusercontent.com/wiki/Konard/LinksPlatform/%
                ↪ D0%9E-%D1%82%D0%BE%D0%BC%2C-%D0%BA%D0%B0%D0%BA-%D0%B2%D1%81%D1%91-%D0%BD%D0%B0%D1%87
                ↪ %D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>
382         private static readonly string _exampleText =
383             @"([english
                ↪ version](https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))
384
385 Обозначение пустоты, какое оно? Темнота ли это? Там где отсутствие света, отсутствие фотонов
    ↪ (носителей света)? Или это то, что полностью отражает свет? Пустой белый лист бумаги? Там
    ↪ где есть место для нового начала? Разве пустота это не характеристика пространства?
    ↪ Пространство это то, что можно чем-то наполнить?
386
387 [![чёрное пространство, белое
    ↪ пространство](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png
    ↪ ""чёрное пространство, белое пространство"")](https://raw.githubusercontent.com/Konard/Links
    ↪ Platform/master/doc/Intro/1.png)
388
389 Что может быть минимальным рисунком, образом, графикой? Может быть это точка? Это ли простейшая
    ↪ форма? Но есть ли у точки размер? Цвет? Масса? Координаты? Время существования?
390
391 [![чёрное пространство, чёрная
    ↪ точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png
    ↪ ""чёрное пространство, чёрная
    ↪ точка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png)
392
393 А что если повторить? Сделать копию? Создать дубликат? Из одного сделать два? Может это быть
    ↪ так? Инверсия? Отражение? Сумма?
394
395 [![белая точка, чёрная
    ↪ точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png ""белая
    ↪ точка, чёрная
    ↪ точка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png)
396
397 А что если мы вообразим движение? Нужно ли время? Каким самым коротким будет путь? Что будет
    ↪ если этот путь зафиксировать? Запомнить след? Как две точки становятся линией? Чертой?
    ↪ Гранью? Разделителем? Единицей?
398
399 [![две белые точки, чёрная вертикальная
    ↪ линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png ""две
    ↪ белые точки, чёрная вертикальная
    ↪ линия"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png)
400
401 Можно ли замкнуть движение? Может ли это быть кругом? Можно ли замкнуть время? Или остаётся
    ↪ только спираль? Но что если замкнуть предел? Создать ограничение, разделение? Получится
    ↪ замкнутая область? Полностью отделённая от всего остального? Но что это всё остальное? Что
    ↪ можно делить? В каком направлении? Ничего или всё? Пустота или полнота? Начало или конец?
    ↪ Или может быть это единица и ноль? Дуальность? Противоположность? А что будет с кругом если
    ↪ у него нет размера? Будет ли круг точкой? Точка состоящая из точек?
402
403 [![белая вертикальная линия, чёрный
    ↪ круг](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png ""белая
    ↪ вертикальная линия, чёрный
    ↪ круг"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png)
404
```

```
Как ещё можно использовать грань, черту, линию? А что если она может что-то соединять, может тогда её нужно повернуть? Почему то, что перпендикулярно вертикальному горизонтально? Горизонт? Инвертирует ли это смысл? Что такое смысл? Из чего состоит смысл? Существует ли элементарная единица смысла?

[![белый круг, чёрная горизонтальная линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png ""белый круг, чёрная горизонтальная линия"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png)

Соединять, допустим, а какой смысл в этом есть ещё? Что если помимо смысла ""соединить, связать"", есть ещё и смысл направления ""от начала к концу""? От предка к потомку? От родителя к ребёнку? От общего к частному?

[![белая горизонтальная линия, чёрная горизонтальная стрелка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png ""белая горизонтальная линия, чёрная горизонтальная стрелка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png)

Шаг назад. Возьмём опять отделённую область, которая лишь та же замкнутая линия, что ещё она может представлять собой? Объект? Но в чём его суть? Разве не в том, что у него есть граница, разделяющая внутреннее и внешнее? Допустим связь, стрелка, линия соединяет два объекта, как бы это выглядело?

[![белая связь, чёрная направленная связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png ""белая связь, чёрная направленная связь"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png)

Допустим у нас есть смысл ""связать"" и смысл ""направления"", много ли это нам даёт? Много ли вариантов интерпретации? А что если уточнить, каким именно образом выполнена связь? Что если можно задать ей чёткий, конкретный смысл? Что это будет? Тип? Глагол? Связка? Действие? Трансформация? Переход из состояния в состояние? Или всё это и есть объект, суть которого в его конечном состоянии, если конечно конец определён направлением?

[![белая обычная и направленная связи, чёрная типизированная связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png ""белая обычная и направленная связи, чёрная типизированная связь"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png)

А что если всё это время, мы смотрели на суть как бы снаружи? Можно ли взглянуть на это изнутри? Что будет внутри объектов? Объекты ли это? Или это связи? Может ли эта структура описать сама себя? Но что тогда получится, разве это не рекурсия? Может это фрактал?

[![белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная типизированная связь с рекурсивной внутренней структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png ""белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная типизированная связь с рекурсивной внутренней структурой"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png)

На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом рекурсии или фрактала?

[![белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная типизированная связь с двойной рекурсивной внутренней структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png ""белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная типизированная связь с двойной рекурсивной внутренней структурой"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png)

Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы? Буквы? Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?

[![белая обычная и направленная связи со структурой из 8 цветных элементов последовательности, чёрная типизированная связь со структурой из 8 цветных элементов последовательности](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png ""белая обычная и направленная связи со структурой из 8 цветных элементов последовательности, чёрная типизированная связь со структурой из 8 цветных элементов последовательности"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png)

...

[![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animation-500.gif ""анимация"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animation-500.gif)";


```

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

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

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

```
626
627                 var duplicateProvider1 = new
                    ↪  DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
628                 var duplicateProvider2 = new
                    ↪  DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
629                 var duplicateProvider3 = new
                    ↪  DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
630
631                 var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
632                 var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
633                 var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
634
635                 var duplicates1 = duplicateCounter1.Count();
636
637                 ConsoleHelpers.Debug("------");
638
639                 var duplicates2 = duplicateCounter2.Count();
640
641                 ConsoleHelpers.Debug("------");
642
643                 var duplicates3 = duplicateCounter3.Count();
644
645                 Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
646
647                 linkFrequenciesCache1.ValidateFrequencies();
648                 linkFrequenciesCache3.ValidateFrequencies();
649             }
650         }
651
652         [Fact]
653         public static void CompressionStabilityTest()
654         {
655             // TODO: Fix bug (do a separate test)
656             //const ulong minNumbers = 0;
657             //const ulong maxNumbers = 1000;
658
659             const ulong minNumbers = 10000;
660             const ulong maxNumbers = 12500;
661
662             var strings = new List<string>();
663
664             for (ulong i = minNumbers; i < maxNumbers; i++)
665             {
666                 strings.Add(i.ToString());
667             }
668
669             var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
670             var totalCharacters = arrays.Select(x => x.Length).Sum();
671
672             using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
                    ↪  SequencesOptions<ulong> { UseCompression = true,
                    ↪  EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
673             using (var scope2 = new TempLinksTestScope(useSequences: true))
674             {
675                 scope1.Links.UseUnicode();
676                 scope2.Links.UseUnicode();
677
678                 //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
679                 var compressor1 = scope1.Sequences;
680                 var compressor2 = scope2.Sequences;
681
682                 var compressed1 = new ulong[arrays.Length];
683                 var compressed2 = new ulong[arrays.Length];
684
685                 var sw1 = Stopwatch.StartNew();
686
687                 var START = 0;
688                 var END = arrays.Length;
689
690                 // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
691                 // Stability issue starts at 10001 or 11000
692                 //for (int i = START; i < END; i++)
693                 //{
694                 //    var first = compressor1.Compress(arrays[i]);
695                 //    var second = compressor1.Compress(arrays[i]);
696
697                 //    if (first == second)
698                 //        compressed1[i] = first;
699                 //    else
700                 //    {
```

```
                    //        // TODO: Find a solution for this case
                    //    }
                    //}

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

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

                    var elapsed1 = sw1.Elapsed;

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

                    var sw2 = Stopwatch.StartNew();

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

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

                    var elapsed2 = sw2.Elapsed;

                    Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
                    ↪  {elapsed2}");

                    Assert.True(elapsed1 > elapsed2);

                    // Checks
                    for (int i = START; i < END; i++)
                    {
                        var sequence1 = compressed1[i];
                        var sequence2 = compressed2[i];

                        if (sequence1 != _constants.Null && sequence2 != _constants.Null)
                        {
                            var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                            ↪  scope1.Links);

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

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

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

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

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

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

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

```

```
772                 //compressor1.ValidateFrequencies();
773             }
774         }
775 
776         [Fact]
777         public static void RundomNumbersCompressionQualityTest()
778         {
779             const ulong N = 500;
780 
781             //const ulong minNumbers = 10000;
782             //const ulong maxNumbers = 20000;
783 
784             //var strings = new List<string>();
785 
786             //for (ulong i = 0; i < N; i++)
787             //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
                ↪ maxNumbers).ToString());
788 
789             var strings = new List<string>();
790 
791             for (ulong i = 0; i < N; i++)
792             {
793                 strings.Add(RandomHelpers.Default.NextUInt64().ToString());
794             }
795 
796             strings = strings.Distinct().ToList();
797 
798             var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
799             var totalCharacters = arrays.Select(x => x.Length).Sum();
800 
801             using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
                ↪ SequencesOptions<ulong> { UseCompression = true,
                ↪ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
802             using (var scope2 = new TempLinksTestScope(useSequences: true))
803             {
804                 scope1.Links.UseUnicode();
805                 scope2.Links.UseUnicode();
806 
807                 var compressor1 = scope1.Sequences;
808                 var compressor2 = scope2.Sequences;
809 
810                 var compressed1 = new ulong[arrays.Length];
811                 var compressed2 = new ulong[arrays.Length];
812 
813                 var sw1 = Stopwatch.StartNew();
814 
815                 var START = 0;
816                 var END = arrays.Length;
817 
818                 for (int i = START; i < END; i++)
819                 {
820                     compressed1[i] = compressor1.Create(arrays[i]);
821                 }
822 
823                 var elapsed1 = sw1.Elapsed;
824 
825                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
826 
827                 var sw2 = Stopwatch.StartNew();
828 
829                 for (int i = START; i < END; i++)
830                 {
831                     compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
832                 }
833 
834                 var elapsed2 = sw2.Elapsed;
835 
836                 Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
                    ↪ {elapsed2}");
837 
838                 Assert.True(elapsed1 > elapsed2);
839 
840                 // Checks
841                 for (int i = START; i < END; i++)
842                 {
843                     var sequence1 = compressed1[i];
844                     var sequence2 = compressed2[i];
845 
846                     if (sequence1 != _constants.Null && sequence2 != _constants.Null)
847                     {
```

```
848                         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                            ↪  scope1.Links);
849
850                         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                            ↪  scope2.Links);
851
852                         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
853                     }
854                 }
855
856                 Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
857                 Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
858
859                 Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
                    ↪  totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
                    ↪  totalCharacters}");
860
861                 // Can be worse than balanced variant
862                 //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
863
864                 //compressor1.ValidateFrequencies();
865             }
866         }
867
868         [Fact]
869         public static void AllTreeBreakDownAtSequencesCreationBugTest()
870         {
871             // Made out of AllPossibleConnectionsTest test.
872
873             //const long sequenceLength = 5; //100% bug
874             const long sequenceLength = 4; //100% bug
875             //const long sequenceLength = 3; //100% _no_bug (ok)
876
877             using (var scope = new TempLinksTestScope(useSequences: true))
878             {
879                 var links = scope.Links;
880                 var sequences = scope.Sequences;
881
882                 var sequence = new ulong[sequenceLength];
883                 for (var i = 0; i < sequenceLength; i++)
884                 {
885                     sequence[i] = links.Create();
886                 }
887
888                 var createResults = sequences.CreateAllVariants2(sequence);
889
890                 Global.Trash = createResults;
891
892                 for (var i = 0; i < sequenceLength; i++)
893                 {
894                     links.Delete(sequence[i]);
895                 }
896             }
897         }
898
899         [Fact]
900         public static void AllPossibleConnectionsTest()
901         {
902             const long sequenceLength = 5;
903
904             using (var scope = new TempLinksTestScope(useSequences: true))
905             {
906                 var links = scope.Links;
907                 var sequences = scope.Sequences;
908
909                 var sequence = new ulong[sequenceLength];
910                 for (var i = 0; i < sequenceLength; i++)
911                 {
912                     sequence[i] = links.Create();
913                 }
914
915                 var createResults = sequences.CreateAllVariants2(sequence);
916                 var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
917
918                 for (var i = 0; i < 1; i++)
919                 {
920                     var sw1 = Stopwatch.StartNew();
921                     var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
922
```

```
923                     var sw2 = Stopwatch.StartNew();
924                     var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();
925
926                     var sw3 = Stopwatch.StartNew();
927                     var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();
928
929                     var sw4 = Stopwatch.StartNew();
930                     var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();
931
932                     Global.Trash = searchResults3;
933                     Global.Trash = searchResults4; //-V3008
934
935                     var intersection1 = createResults.Intersect(searchResults1).ToList();
936                     Assert.True(intersection1.Count == createResults.Length);
937
938                     var intersection2 = reverseResults.Intersect(searchResults1).ToList();
939                     Assert.True(intersection2.Count == reverseResults.Length);
940
941                     var intersection0 = searchResults1.Intersect(searchResults2).ToList();
942                     Assert.True(intersection0.Count == searchResults2.Count);
943
944                     var intersection3 = searchResults2.Intersect(searchResults3).ToList();
945                     Assert.True(intersection3.Count == searchResults3.Count);
946
947                     var intersection4 = searchResults3.Intersect(searchResults4).ToList();
948                     Assert.True(intersection4.Count == searchResults4.Count);
949                 }
950
951                 for (var i = 0; i < sequenceLength; i++)
952                 {
953                     links.Delete(sequence[i]);
954                 }
955             }
956         }
957
958         [Fact(Skip = "Correct implementation is pending")]
959         public static void CalculateAllUsagesTest()
960         {
961             const long sequenceLength = 3;
962
963             using (var scope = new TempLinksTestScope(useSequences: true))
964             {
965                 var links = scope.Links;
966                 var sequences = scope.Sequences;
967
968                 var sequence = new ulong[sequenceLength];
969                 for (var i = 0; i < sequenceLength; i++)
970                 {
971                     sequence[i] = links.Create();
972                 }
973
974                 var createResults = sequences.CreateAllVariants2(sequence);
975
976                 //var reverseResults =
                    ↪  sequences.CreateAllVariants2(sequence.Reverse().ToArray());
977
978                 for (var i = 0; i < 1; i++)
979                 {
980                     var linksTotalUsages1 = new ulong[links.Count() + 1];
981
982                     sequences.CalculateAllUsages(linksTotalUsages1);
983
984                     var linksTotalUsages2 = new ulong[links.Count() + 1];
985
986                     sequences.CalculateAllUsages2(linksTotalUsages2);
987
988                     var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
989                     Assert.True(intersection1.Count == linksTotalUsages2.Length);
990                 }
991
992                 for (var i = 0; i < sequenceLength; i++)
993                 {
994                     links.Delete(sequence[i]);
995                 }
996             }
997         }
998     }
999 }
```

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

```
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/UnaryNumberConvertersTests.cs

```
using Xunit;
using Platform.Random;
using Platform.Data.Doublets.UnaryNumbers;

namespace Platform.Data.Doublets.Tests
{
    public static class UnaryNumberConvertersTests
    {
        [Fact]
        public static void ConvertersTest()
        {
            using (var scope = new TempLinksTestScope())
            {
                const int N = 10;
                var links = scope.Links;
```

```
                var meaningRoot = links.CreatePoint();
                var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
                var powerOf2ToUnaryNumberConverter = new PowerOf2ToUnaryNumberConverter<ulong>(links, one);
                var toUnaryNumberConverter = new AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
                var random = new System.Random(0);
                ulong[] numbers = new ulong[N];
                ulong[] unaryNumbers = new ulong[N];
                for (int i = 0; i < N; i++)
                {
                    numbers[i] = random.NextUInt64();
                    unaryNumbers[i] = toUnaryNumberConverter.Convert(numbers[i]);
                }
                var fromUnaryNumberConverterUsingOrOperation = new UnaryNumberToAddressOrOperationConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
                var fromUnaryNumberConverterUsingAddOperation = new UnaryNumberToAddressAddOperationConverter<ulong>(links, one);
                for (int i = 0; i < N; i++)
                {
                    Assert.Equal(numbers[i], fromUnaryNumberConverterUsingOrOperation.Convert(unaryNumbers[i]));
                    Assert.Equal(numbers[i], fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));
                }
            }
        }
    }
}
```

./Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs

```
using Platform.Data.Doublets.Incrementers;
using Platform.Data.Doublets.PropertyOperators;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Sequences.Indexes;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.UnaryNumbers;
using Platform.Data.Doublets.Unicode;
using Xunit;

namespace Platform.Data.Doublets.Tests
{
    public static class UnicodeConvertersTests
    {
        [Fact]
        public static void CharAndUnicodeSymbolConvertersTest()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;

                var itself = links.Constants.Itself;

                var meaningRoot = links.CreatePoint();
                var one = links.CreateAndUpdate(meaningRoot, itself);
                var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);

                var powerOf2ToUnaryNumberConverter = new PowerOf2ToUnaryNumberConverter<ulong>(links, one);
                var addressToUnaryNumberConverter = new AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
                var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter, unicodeSymbolMarker);

                var originalCharacter = 'H';

                var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);

                var unaryNumberToAddressConverter = new UnaryNumberToAddressOrOperationConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
                var unicodeSymbolCriterionMatcher = new UnicodeSymbolCriterionMatcher<ulong>(links, unicodeSymbolMarker);
                var unicodeSymbolToCharConverter = new UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter, unicodeSymbolCriterionMatcher);

```

```
39                  var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
40
41                  Assert.Equal(originalCharacter, resultingCharacter);
42              }
43          }
44
45          [Fact]
46          public static void StringAndUnicodeSequenceConvertersTest()
47          {
48              using (var scope = new TempLinksTestScope())
49              {
50                  var links = scope.Links;
51
52                  var itself = links.Constants.Itself;
53
54                  var meaningRoot = links.CreatePoint();
55                  var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
56                  var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
57                  var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
58                  var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
59                  var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);
60
61                  var powerOf2ToUnaryNumberConverter = new
                    ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
62                  var addressToUnaryNumberConverter = new
                    ↪  AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
63                  var charToUnicodeSymbolConverter = new
                    ↪  CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
                    ↪  unicodeSymbolMarker);
64
65                  var unaryNumberToAddressConverter = new
                    ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links,
                    ↪  powerOf2ToUnaryNumberConverter);
66                  var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
67                  var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                    ↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
68                  var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
                    ↪  frequencyPropertyMarker, frequencyMarker);
69                  var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
                    ↪  frequencyPropertyOperator, frequencyIncrementer);
70                  var linkToItsFrequencyNumberConverter = new
                    ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                    ↪  unaryNumberToAddressConverter);
71                  var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                    ↪  linkToItsFrequencyNumberConverter);
72                  var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                    ↪  sequenceToItsLocalElementLevelsConverter);
73
74                  var stringToUnicodeSymbolConverter = new
                    ↪  StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
                    ↪  index, optimalVariantConverter, unicodeSequenceMarker);
75
76                  var originalString = "Hello";
77
78                  var unicodeSequenceLink = stringToUnicodeSymbolConverter.Convert(originalString);
79
80                  var unicodeSymbolCriterionMatcher = new
                    ↪  UnicodeSymbolCriterionMatcher<ulong>(links, unicodeSymbolMarker);
81                  var unicodeSymbolToCharConverter = new
                    ↪  UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
                    ↪  unicodeSymbolCriterionMatcher);
82
83                  var unicodeSequenceCriterionMatcher = new
                    ↪  UnicodeSequenceCriterionMatcher<ulong>(links, unicodeSequenceMarker);
84
85                  var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
                    ↪  unicodeSymbolCriterionMatcher.IsMatched);
86
87                  var unicodeSequenceToStringConverter = new
                    ↪  UnicodeSequenceToStringConverter<ulong>(links,
                    ↪  unicodeSequenceCriterionMatcher, sequenceWalker,
                    ↪  unicodeSymbolToCharConverter);
88
89                  var resultingString =
                    ↪  unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
90
91                  Assert.Equal(originalString, resultingString);
```

```
                }
            }
        }
    }
```

Index