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

```
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

```
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;

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;

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;

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;

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;

    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;

    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;

    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;

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;

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

```
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;

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;

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;
            public IList<TLink> After;

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

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

        // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution
        ↪  (Links-Expression)
        public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>, TLink>
        ↪  matchedHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
        ↪  substitutedHandler)
        {
            ////List<Transition> transitions = null;
            ////if (!restriction.IsNullOrEmpty())
            ////{
            ////    // Есть причина делать проход (чтение)
```

```
////    if (matchedHandler != null)
////    {
////        if (!substitution.IsNullOrEmpty())
////        {
////            // restriction => { 0, 0, 0 } | { 0 } // Create
////            // substitution => { itself, 0, 0 } | { itself, itself, itself } // Create / Update
////            // substitution => { 0, 0, 0 } | { 0 } // Delete
////            transitions = new List<Transition>();
////            if (Equals(substitution[Constants.IndexPart], Constants.Null))
////            {
////                // If index is Null, that means we always ignore every other value (they are also Null by definition)
////                var matchDecision = matchedHandler(, NullLink);
////                if (Equals(matchDecision, Constants.Break))
////                    return false;
////                if (!Equals(matchDecision, Constants.Skip))
////                    transitions.Add(new Transition(matchedLink, newValue));
////            }
////            else
////            {
////                Func<T, bool> handler;
////                handler = link =>
////                {
////                    var matchedLink = Memory.GetLinkValue(link);
////                    var newValue = Memory.GetLinkValue(link);
////                    newValue[Constants.IndexPart] = Constants.Itself;
////                    newValue[Constants.SourcePart] = Equals(substitution[Constants.SourcePart], Constants.Itself) ? matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
////                    newValue[Constants.TargetPart] = Equals(substitution[Constants.TargetPart], Constants.Itself) ? matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
////                    var matchDecision = matchedHandler(matchedLink, newValue);
////                    if (Equals(matchDecision, Constants.Break))
////                        return false;
////                    if (!Equals(matchDecision, Constants.Skip))
////                        transitions.Add(new Transition(matchedLink, newValue));
////                    return true;
////                };
////                if (!Memory.Each(handler, restriction))
////                    return Constants.Break;
////            }
////        }
////        else
////        {
////            Func<T, bool> handler = link =>
////            {
////                var matchedLink = Memory.GetLinkValue(link);
////                var matchDecision = matchedHandler(matchedLink, matchedLink);
////                return !Equals(matchDecision, Constants.Break);
////            };
////            if (!Memory.Each(handler, restriction))
////                return Constants.Break;
////        }
////    }
////    else
////    {
////        if (substitution != null)
////        {
////            transitions = new List<IList<T>>();
////            Func<T, bool> handler = link =>
////            {
////                var matchedLink = Memory.GetLinkValue(link);
////                transitions.Add(matchedLink);
////                return true;
////            };
////            if (!Memory.Each(handler, restriction))
////                return Constants.Break;
////        }
////        else
////        {
////            return Constants.Continue;
////        }
////    }
////}
////if (substitution != null)
```

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

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

```
189             return Constants.Continue;
190         }
191 
192         public TLink Trigger(IList<TLink> patternOrCondition, Func<IList<TLink>, TLink>
            ↪ matchHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
            ↪ substitutionHandler)
193         {
194             if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
195             {
196                 return Constants.Continue;
197             }
198             else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO:
                ↪ Check if it is a correct condition
199             {
200                 // Or it only applies to trigger without matchHandler.
201                 throw new NotImplementedException();
202             }
203             else if (!substitution.IsNullOrEmpty()) // Creation
204             {
205                 var before = ArrayPool<TLink>.Empty;
206                 // Что должно означать False здесь? Остановиться (перестать идти) или пропустить
                    ↪ (пройти мимо) или пустить (взять)?
207                 if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                    ↪ Constants.Break))
208                 {
209                     return Constants.Break;
210                 }
211                 var after = (IList<TLink>)substitution.ToArray();
212                 if (_equalityComparer.Equals(after[0], default))
213                 {
214                     var newLink = Links.Create();
215                     after[0] = newLink;
216                 }
217                 if (substitution.Count == 1)
218                 {
219                     after = Links.GetLink(substitution[0]);
220                 }
221                 else if (substitution.Count == 3)
222                 {
223                     Links.Update(after);
224                 }
225                 else
226                 {
227                     throw new NotSupportedException();
228                 }
229                 if (matchHandler != null)
230                 {
231                     return substitutionHandler(before, after);
232                 }
233                 return Constants.Continue;
234             }
235             else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
236             {
237                 if (patternOrCondition.Count == 1)
238                 {
239                     var linkToDelete = patternOrCondition[0];
240                     var before = Links.GetLink(linkToDelete);
241                     if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                        ↪ Constants.Break))
242                     {
243                         return Constants.Break;
244                     }
245                     var after = ArrayPool<TLink>.Empty;
246                     Links.Update(linkToDelete, Constants.Null, Constants.Null);
247                     Links.Delete(linkToDelete);
248                     if (matchHandler != null)
249                     {
250                         return substitutionHandler(before, after);
251                     }
252                     return Constants.Continue;
253                 }
254                 else
255                 {
256                     throw new NotSupportedException();
257                 }
258             }
259             else // Replace / Update
260             {
```

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

        /// <remarks>
        /// IList[IList[IList[T]]]
        /// |      |      |       |||
        /// |      |      |       ------ ||
        /// |      |      |        link  ||
        /// |      |      -------------- |
        /// |      |         change      |
        ///  --------------------
        ///         changes
        /// </remarks>
        public IList<IList<IList<TLink>>> Trigger(IList<TLink> condition, IList<TLink>
        ↪  substitution)
        {
            var changes = new List<IList<IList<TLink>>>();
            Trigger(condition, AlwaysContinue, substitution, (before, after) =>
            {
                var change = new[] { before, after };
                changes.Add(change);
                return Constants.Continue;
            });
            return changes;
        }

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

./Platform.Data.Doublets/DoubletComparer.cs

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

namespace Platform.Data.Doublets
{
    /// <remarks>
```

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

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

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

./Platform.Data.Doublets/Doublet.cs

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

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

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

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

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

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

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

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

./Platform.Data.Doublets/Hybrid.cs

```
using System;
using System.Reflection;
using Platform.Reflection;
using Platform.Converters;
using Platform.Exceptions;

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

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

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

        public Hybrid(object value, bool isExternal)
        {
            var signedType = Type<T>.SignedVersion;
            var signedValue = Convert.ChangeType(value, signedType);
            var abs = typeof(Numbers.Math).GetTypeInfo().GetMethod("Abs").MakeGenericMethod(sign
            ↪ edType);
```

```
            var negate = typeof(Numbers.Math).GetTypeInfo().GetMethod("Negate").MakeGenericMethod(signedType);
            var absoluteValue = abs.Invoke(null, new[] { signedValue });
            var resultValue = isExternal ? negate.Invoke(null, new[] { absoluteValue }) : absoluteValue;
            Value = To.UnsignedAs<T>(resultValue);
        }

        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;

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;

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

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

```
89                  throw new Exception("В процессе поиска по хранилищу не было найдено связей.");
90              }
91              return firstLink;
92          }
93
94          public static bool IsInnerReference<TLink>(this ILinks<TLink> links, TLink reference)
95          {
96              var constants = links.Constants;
97              var comparer = Comparer<TLink>.Default;
98              return comparer.Compare(constants.MinPossibleIndex, reference) >= 0 &&
                ↪  comparer.Compare(reference, constants.MaxPossibleIndex) <= 0;
99          }
100
101         #region Paths
102
103         /// <remarks>
104         /// TODO: Как так? Как то что ниже может быть корректно?
105         /// Скорее всего практически не применимо
106         /// Предполагалось, что можно было конвертировать формируемый в проходе через
            ↪  SequenceWalker
107         /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
108         /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
109         /// </remarks>
110         public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[]
            ↪  path)
111         {
112             var current = path[0];
113             //EnsureLinkExists(current, "path");
114             if (!links.Exists(current))
115             {
116                 return false;
117             }
118             var equalityComparer = EqualityComparer<TLink>.Default;
119             var constants = links.Constants;
120             for (var i = 1; i < path.Length; i++)
121             {
122                 var next = path[i];
123                 var values = links.GetLink(current);
124                 var source = values[constants.SourcePart];
125                 var target = values[constants.TargetPart];
126                 if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
                    ↪  next))
127                 {
128                     //throw new Exception(string.Format("Невозможно выбрать путь, так как и
                        ↪  Source и Target совпадают с элементом пути {0}.", next));
129                     return false;
130                 }
131                 if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
                    ↪  target))
132                 {
133                     //throw new Exception(string.Format("Невозможно продолжить путь через
                        ↪  элемент пути {0}", next));
134                     return false;
135                 }
136                 current = next;
137             }
138             return true;
139         }
140
141         /// <remarks>
142         /// Может потребовать дополнительного стека для PathElement's при использовании
            ↪  SequenceWalker.
143         /// </remarks>
144         public static TLink GetByKeys<TLink>(this ILinks<TLink> links, TLink root, params int[]
            ↪  path)
145         {
146             links.EnsureLinkExists(root, "root");
147             var currentLink = root;
148             for (var i = 0; i < path.Length; i++)
149             {
150                 currentLink = links.GetLink(currentLink)[path[i]];
151             }
152             return currentLink;
153         }
154
155         public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this ILinks<TLink>
            ↪  links, TLink root, ulong size, ulong index)
156         {
```

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

```
225         /// <param name="links">Хранилище связей.</param>
226         /// <param name="handler">Обработчик каждой подходящей связи.</param>
227         /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
            ↪ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
            ↪ Any - отсутствие ограничения, 1..∞ конкретный адрес связи.</param>
228         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪ случае.</returns>
229         [MethodImpl(MethodImplOptions.AggressiveInlining)]
230         public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
            ↪ handler, params TLink[] restrictions)
231             => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
                ↪ links.Constants.Continue);
232
233         /// <summary>
234         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪ (handler) для каждой подходящей связи.
235         /// </summary>
236         /// <param name="links">Хранилище связей.</param>
237         /// <param name="source">Значение, определяющее соответствующие шаблону связи.
            ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
            ↪ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
238         /// <param name="target">Значение, определяющее соответствующие шаблону связи.
            ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
            ↪ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
239         /// <param name="handler">Обработчик каждой подходящей связи.</param>
240         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪ случае.</returns>
241         [MethodImpl(MethodImplOptions.AggressiveInlining)]
242         public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
            ↪ Func<TLink, bool> handler)
243         {
244             var constants = links.Constants;
245             return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
                ↪ constants.Break, constants.Any, source, target);
246         }
247
248         /// <summary>
249         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪ (handler) для каждой подходящей связи.
250         /// </summary>
251         /// <param name="links">Хранилище связей.</param>
252         /// <param name="source">Значение, определяющее соответствующие шаблону связи.
            ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
            ↪ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
253         /// <param name="target">Значение, определяющее соответствующие шаблону связи.
            ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
            ↪ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
254         /// <param name="handler">Обработчик каждой подходящей связи.</param>
255         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪ случае.</returns>
256         [MethodImpl(MethodImplOptions.AggressiveInlining)]
257         public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
            ↪ Func<IList<TLink>, TLink> handler)
258         {
259             var constants = links.Constants;
260             return links.Each(handler, constants.Any, source, target);
261         }
262
263         [MethodImpl(MethodImplOptions.AggressiveInlining)]
264         public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
            ↪ restrictions)
265         {
266             long arraySize = (Integer<TLink>)links.Count(restrictions);
267             var array = new IList<TLink>[arraySize];
268             if (arraySize > 0)
269             {
270                 var filler = new ArrayFiller<IList<TLink>, TLink>(array,
                    ↪ links.Constants.Continue);
271                 links.Each(filler.AddAndReturnConstant, restrictions);
272             }
273             return array;
274         }
275
276         [MethodImpl(MethodImplOptions.AggressiveInlining)]
277         public static IList<TLink> AllIndices<TLink>(this ILinks<TLink> links, params TLink[]
            ↪ restrictions)
```

```
278         {
279             long arraySize = (Integer<TLink>)links.Count(restrictions);
280             var array = new TLink[arraySize];
281             if (arraySize > 0)
282             {
283                 var filler = new ArrayFiller<TLink, TLink>(array, links.Constants.Continue);
284                 links.Each(filler.AddFirstAndReturnConstant, restrictions);
285             }
286             return array;
287         }
288 
289         /// <summary>
290         /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
            ↪ в хранилище связей.
291         /// </summary>
292         /// <param name="links">Хранилище связей.</param>
293         /// <param name="source">Начало связи.</param>
294         /// <param name="target">Конец связи.</param>
295         /// <returns>Значение, определяющее существует ли связь.</returns>
296         [MethodImpl(MethodImplOptions.AggressiveInlining)]
297         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;
298 
299         #region Ensure
300         // TODO: May be move to EnsureExtensions or make it both there and here
301 
302         [MethodImpl(MethodImplOptions.AggressiveInlining)]
303         public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
            ↪ reference, string argumentName)
304         {
305             if (links.IsInnerReference(reference) && !links.Exists(reference))
306             {
307                 throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
308             }
309         }
310 
311         [MethodImpl(MethodImplOptions.AggressiveInlining)]
312         public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
            ↪ IList<TLink> restrictions, string argumentName)
313         {
314             for (int i = 0; i < restrictions.Count; i++)
315             {
316                 links.EnsureInnerReferenceExists(restrictions[i], argumentName);
317             }
318         }
319 
320         [MethodImpl(MethodImplOptions.AggressiveInlining)]
321         public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
            ↪ restrictions)
322         {
323             for (int i = 0; i < restrictions.Count; i++)
324             {
325                 links.EnsureLinkIsAnyOrExists(restrictions[i], nameof(restrictions));
326             }
327         }
328 
329         [MethodImpl(MethodImplOptions.AggressiveInlining)]
330         public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
            ↪ string argumentName)
331         {
332             var equalityComparer = EqualityComparer<TLink>.Default;
333             if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
334             {
335                 throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
336             }
337         }
338 
339         [MethodImpl(MethodImplOptions.AggressiveInlining)]
340         public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links, TLink
            ↪ link, string argumentName)
341         {
342             var equalityComparer = EqualityComparer<TLink>.Default;
343             if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
344             {
345                 throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
346             }
347         }
```

```
348
349         /// <param name="links">Хранилище связей.</param>
350         [MethodImpl(MethodImplOptions.AggressiveInlining)]
351         public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink source,
            ↪ TLink target)
352         {
353             if (links.Exists(source, target))
354             {
355                 throw new LinkWithSameValueAlreadyExistsException();
356             }
357         }
358
359         /// <param name="links">Хранилище связей.</param>
360         public static void EnsureNoUsages<TLink>(this ILinks<TLink> links, TLink link)
361         {
362             if (links.HasUsages(link))
363             {
364                 throw new ArgumentLinkHasDependenciesException<TLink>(link);
365             }
366         }
367
368         /// <param name="links">Хранилище связей.</param>
369         public static void EnsureCreated<TLink>(this ILinks<TLink> links, params TLink[]
            ↪ addresses) => links.EnsureCreated(links.Create, addresses);
370
371         /// <param name="links">Хранилище связей.</param>
372         public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params TLink[]
            ↪ addresses) => links.EnsureCreated(links.CreatePoint, addresses);
373
374         /// <param name="links">Хранилище связей.</param>
375         public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
            ↪ params TLink[] addresses)
376         {
377             var constants = links.Constants;
378             var nonExistentAddresses = new HashSet<ulong>(addresses.Where(x =>
                ↪ !links.Exists(x)).Select(x => (ulong)(Integer<TLink>)x));
379             if (nonExistentAddresses.Count > 0)
380             {
381                 var max = nonExistentAddresses.Max();
382                 // TODO: Эту верхнюю границу нужно разрешить переопределять (проверить
                    ↪ применяется ли эта логика)
383                 max = System.Math.Min(max, (Integer<TLink>)constants.MaxPossibleIndex);
384                 var createdLinks = new List<TLink>();
385                 var equalityComparer = EqualityComparer<TLink>.Default;
386                 TLink createdLink = creator();
387                 while (!equalityComparer.Equals(createdLink, (Integer<TLink>)max))
388                 {
389                     createdLinks.Add(createdLink);
390                 }
391                 for (var i = 0; i < createdLinks.Count; i++)
392                 {
393                     if (!nonExistentAddresses.Contains((Integer<TLink>)createdLinks[i]))
394                     {
395                         links.Delete(createdLinks[i]);
396                     }
397                 }
398             }
399         }
400
401         #endregion
402
403         /// <param name="links">Хранилище связей.</param>
404         public static ulong CountUsages<TLink>(this ILinks<TLink> links, TLink link)
405         {
406             var constants = links.Constants;
407             var values = links.GetLink(link);
408             ulong usagesAsSource = (Integer<TLink>)links.Count(new Link<TLink>(constants.Any,
                ↪ link, constants.Any));
409             var equalityComparer = EqualityComparer<TLink>.Default;
410             if (equalityComparer.Equals(values[constants.SourcePart], link))
411             {
412                 usagesAsSource--;
413             }
414             ulong usagesAsTarget = (Integer<TLink>)links.Count(new Link<TLink>(constants.Any,
                ↪ constants.Any, link));
415             if (equalityComparer.Equals(values[constants.TargetPart], link))
416             {
417                 usagesAsTarget--;
```

```
418             }
419             return usagesAsSource + usagesAsTarget;
420         }
421 
422         /// <param name="links">Хранилище связей.</param>
423         [MethodImpl(MethodImplOptions.AggressiveInlining)]
424         public static bool HasUsages<TLink>(this ILinks<TLink> links, TLink link) =>
            ↪ links.CountUsages(link) > 0;
425 
426         /// <param name="links">Хранилище связей.</param>
427         [MethodImpl(MethodImplOptions.AggressiveInlining)]
428         public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source,
            ↪ TLink target)
429         {
430             var constants = links.Constants;
431             var values = links.GetLink(link);
432             var equalityComparer = EqualityComparer<TLink>.Default;
433             return equalityComparer.Equals(values[constants.SourcePart], source) &&
                ↪ equalityComparer.Equals(values[constants.TargetPart], target);
434         }
435 
436         /// <summary>
437         /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
438         /// </summary>
439         /// <param name="links">Хранилище связей.</param>
440         /// <param name="source">Индекс связи, которая является началом для искомой
            ↪ связи.</param>
441         /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
442         /// <returns>Индекс искомой связи с указанными Source (началом) и Target
            ↪ (концом).</returns>
443         [MethodImpl(MethodImplOptions.AggressiveInlining)]
444         public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
            ↪ target)
445         {
446             var contants = links.Constants;
447             var setter = new Setter<TLink, TLink>(contants.Continue, contants.Break, default);
448             links.Each(setter.SetFirstAndReturnFalse, contants.Any, source, target);
449             return setter.Result;
450         }
451 
452         /// <param name="links">Хранилище связей.</param>
453         [MethodImpl(MethodImplOptions.AggressiveInlining)]
454         public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
455         {
456             var link = links.Create();
457             return links.Update(link, link, link);
458         }
459 
460         /// <param name="links">Хранилище связей.</param>
461         [MethodImpl(MethodImplOptions.AggressiveInlining)]
462         public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
            ↪ target) => links.Update(links.Create(), source, target);
463 
464         /// <summary>
465         /// Обновляет связь с указанными началом (Source) и концом (Target)
466         /// на связь с указанными началом (NewSource) и концом (NewTarget).
467         /// </summary>
468         /// <param name="links">Хранилище связей.</param>
469         /// <param name="link">Индекс обновляемой связи.</param>
470         /// <param name="newSource">Индекс связи, которая является началом связи, на которую
            ↪ выполняется обновление.</param>
471         /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
            ↪ выполняется обновление.</param>
472         /// <returns>Индекс обновлённой связи.</returns>
473         [MethodImpl(MethodImplOptions.AggressiveInlining)]
474         public static TLink Update<TLink>(this ILinks<TLink> links, TLink link, TLink newSource,
            ↪ TLink newTarget) => links.Update(new Link<TLink>(link, newSource, newTarget));
475 
476         /// <summary>
477         /// Обновляет связь с указанными началом (Source) и концом (Target)
478         /// на связь с указанными началом (NewSource) и концом (NewTarget).
479         /// </summary>
480         /// <param name="links">Хранилище связей.</param>
481         /// <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>
482         /// <returns>Индекс обновлённой связи.</returns>
```

```
483             [MethodImpl(MethodImplOptions.AggressiveInlining)]
484             public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
485             {
486                 if (restrictions.Length == 2)
487                 {
488                     return links.MergeAndDelete(restrictions[0], restrictions[1]);
489                 }
490                 if (restrictions.Length == 4)
491                 {
492                     return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
                        ↪  restrictions[2], restrictions[3]);
493                 }
494                 else
495                 {
496                     return links.Update(restrictions);
497                 }
498             }
499 
500             [MethodImpl(MethodImplOptions.AggressiveInlining)]
501             public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
                ↪  links, TLink constant, IList<TLink> restrictions)
502             {
503                 var equalityComparer = EqualityComparer<TLink>.Default;
504                 var constants = links.Constants;
505                 var index = restrictions[constants.IndexPart];
506                 var source = restrictions[constants.SourcePart];
507                 var target = restrictions[constants.TargetPart];
508                 source = equalityComparer.Equals(source, constant) ? index : source;
509                 target = equalityComparer.Equals(target, constant) ? index : target;
510                 return new Link<TLink>(index, source, target);
511             }
512 
513             /// <summary>
514             /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
                ↪  с указанными Source (началом) и Target (концом).
515             /// </summary>
516             /// <param name="links">Хранилище связей.</param>
517             /// <param name="source">Индекс связи, которая является началом на создаваемой
                ↪  связи.</param>
518             /// <param name="target">Индекс связи, которая является концом для создаваемой
                ↪  связи.</param>
519             /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
520             [MethodImpl(MethodImplOptions.AggressiveInlining)]
521             public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
                ↪  target)
522             {
523                 var link = links.SearchOrDefault(source, target);
524                 if (EqualityComparer<TLink>.Default.Equals(link, default))
525                 {
526                     link = links.CreateAndUpdate(source, target);
527                 }
528                 return link;
529             }
530 
531             /// <summary>
532             /// Обновляет связь с указанными началом (Source) и концом (Target)
533             /// на связь с указанными началом (NewSource) и концом (NewTarget).
534             /// </summary>
535             /// <param name="links">Хранилище связей.</param>
536             /// <param name="source">Индекс связи, которая является началом обновляемой
                ↪  связи.</param>
537             /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
538             /// <param name="newSource">Индекс связи, которая является началом связи, на которую
                ↪  выполняется обновление.</param>
539             /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
                ↪  выполняется обновление.</param>
540             /// <returns>Индекс обновлённой связи.</returns>
541             [MethodImpl(MethodImplOptions.AggressiveInlining)]
542             public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
                ↪  TLink target, TLink newSource, TLink newTarget)
543             {
544                 var equalityComparer = EqualityComparer<TLink>.Default;
545                 var link = links.SearchOrDefault(source, target);
546                 if (equalityComparer.Equals(link, default))
547                 {
548                     return links.CreateAndUpdate(newSource, newTarget);
549                 }
```

```
            if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
            ↪  target))
            {
                return link;
            }
            return links.Update(link, newSource, newTarget);
        }

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

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

        /// <remarks>Before execution of this method ensure that deleted link is detached (all
        ↪  values - source and target are reset to null) or it might enter into infinite
        ↪  recursion.</remarks>
        public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
        {
            var anyConstant = links.Constants.Any;
            var usagesAsSourceQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
            links.DeleteByQuery(usagesAsSourceQuery);
            var usagesAsTargetQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
            links.DeleteByQuery(usagesAsTargetQuery);
        }

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

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

```
623             return true;
624         }
625 
626         // TODO: Create a universal version of this method in Platform.Data (with using of for
        ↪  loop)
627         public static void ResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
628         {
629             var nullConstant = links.Constants.Null;
630             var updateRequest = new Link<TLink>(linkIndex, nullConstant, nullConstant);
631             links.Update(updateRequest);
632         }
633 
634         // TODO: Create a universal version of this method in Platform.Data (with using of for
        ↪  loop)
635         public static void EnforceResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
636         {
637             if (!links.AreValuesReset(linkIndex))
638             {
639                 links.ResetValues(linkIndex);
640             }
641         }
642 
643         /// <summary>
644         /// Merging two usages graphs, all children of old link moved to be children of new link
        ↪  or deleted.
645         /// </summary>
646         public static TLink MergeUsages<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
        ↪  TLink newLinkIndex)
647         {
648             var equalityComparer = EqualityComparer<TLink>.Default;
649             if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
650             {
651                 var constants = links.Constants;
652                 var usagesAsSourceQuery = new Link<TLink>(constants.Any, oldLinkIndex,
                ↪  constants.Any);
653                 long usagesAsSourceCount = (Integer<TLink>)links.Count(usagesAsSourceQuery);
654                 var usagesAsTargetQuery = new Link<TLink>(constants.Any, constants.Any,
                ↪  oldLinkIndex);
655                 long usagesAsTargetCount = (Integer<TLink>)links.Count(usagesAsTargetQuery);
656                 var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(oldLinkIndex)) &&
                ↪  usagesAsSourceCount == 1 && usagesAsTargetCount == 1;
657                 if (!isStandalonePoint)
658                 {
659                     var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
660                     if (totalUsages > 0)
661                     {
662                         var usages = ArrayPool.Allocate<TLink>(totalUsages);
663                         var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
                        ↪  links.Constants.Continue);
664                         var i = 0L;
665                         if (usagesAsSourceCount > 0)
666                         {
667                             links.Each(usagesFiller.AddFirstAndReturnConstant,
                            ↪  usagesAsSourceQuery);
668                             for (; i < usagesAsSourceCount; i++)
669                             {
670                                 var usage = usages[i];
671                                 if (!equalityComparer.Equals(usage, oldLinkIndex))
672                                 {
673                                     links.Update(usage, newLinkIndex, links.GetTarget(usage));
674                                 }
675                             }
676                         }
677                         if (usagesAsTargetCount > 0)
678                         {
679                             links.Each(usagesFiller.AddFirstAndReturnConstant,
                            ↪  usagesAsTargetQuery);
680                             for (; i < usages.Length; i++)
681                             {
682                                 var usage = usages[i];
683                                 if (!equalityComparer.Equals(usage, oldLinkIndex))
684                                 {
685                                     links.Update(usage, links.GetSource(usage), newLinkIndex);
686                                 }
687                             }
688                         }
689                         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;

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;

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;

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;

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;
        }

```

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

```
        }

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

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

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

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

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

        public int IndexOf(TLink item)
        {
            if (_equalityComparer.Equals(Index, item))
            {
                return _constants.IndexPart;
            }
            if (_equalityComparer.Equals(Source, item))
            {
                return _constants.SourcePart;
            }
            if (_equalityComparer.Equals(Target, item))
            {
                return _constants.TargetPart;
            }
            return -1;
        }

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

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

        #endregion
    }
}
```

./Platform.Data.Doublets/LinkExtensions.cs

```
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

```
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;

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;

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

// 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();
```

```
46              public static readonly int RightAsTargetOffset = Marshal.OffsetOf(typeof(Link),
                ↪  nameof(RightAsTarget)).ToInt32();
47              public static readonly int SizeAsTargetOffset = Marshal.OffsetOf(typeof(Link),
                ↪  nameof(SizeAsTarget)).ToInt32();
48
49              public TLink Source;
50              public TLink Target;
51              public TLink LeftAsSource;
52              public TLink RightAsSource;
53              public TLink SizeAsSource;
54              public TLink LeftAsTarget;
55              public TLink RightAsTarget;
56              public TLink SizeAsTarget;
57
58              [MethodImpl(MethodImplOptions.AggressiveInlining)]
59              public static TLink GetSource(IntPtr pointer) => (pointer +
                ↪  SourceOffset).GetValue<TLink>();
60              [MethodImpl(MethodImplOptions.AggressiveInlining)]
61              public static TLink GetTarget(IntPtr pointer) => (pointer +
                ↪  TargetOffset).GetValue<TLink>();
62              [MethodImpl(MethodImplOptions.AggressiveInlining)]
63              public static TLink GetLeftAsSource(IntPtr pointer) => (pointer +
                ↪  LeftAsSourceOffset).GetValue<TLink>();
64              [MethodImpl(MethodImplOptions.AggressiveInlining)]
65              public static TLink GetRightAsSource(IntPtr pointer) => (pointer +
                ↪  RightAsSourceOffset).GetValue<TLink>();
66              [MethodImpl(MethodImplOptions.AggressiveInlining)]
67              public static TLink GetSizeAsSource(IntPtr pointer) => (pointer +
                ↪  SizeAsSourceOffset).GetValue<TLink>();
68              [MethodImpl(MethodImplOptions.AggressiveInlining)]
69              public static TLink GetLeftAsTarget(IntPtr pointer) => (pointer +
                ↪  LeftAsTargetOffset).GetValue<TLink>();
70              [MethodImpl(MethodImplOptions.AggressiveInlining)]
71              public static TLink GetRightAsTarget(IntPtr pointer) => (pointer +
                ↪  RightAsTargetOffset).GetValue<TLink>();
72              [MethodImpl(MethodImplOptions.AggressiveInlining)]
73              public static TLink GetSizeAsTarget(IntPtr pointer) => (pointer +
                ↪  SizeAsTargetOffset).GetValue<TLink>();
74
75              [MethodImpl(MethodImplOptions.AggressiveInlining)]
76              public static void SetSource(IntPtr pointer, TLink value) => (pointer +
                ↪  SourceOffset).SetValue(value);
77              [MethodImpl(MethodImplOptions.AggressiveInlining)]
78              public static void SetTarget(IntPtr pointer, TLink value) => (pointer +
                ↪  TargetOffset).SetValue(value);
79              [MethodImpl(MethodImplOptions.AggressiveInlining)]
80              public static void SetLeftAsSource(IntPtr pointer, TLink value) => (pointer +
                ↪  LeftAsSourceOffset).SetValue(value);
81              [MethodImpl(MethodImplOptions.AggressiveInlining)]
82              public static void SetRightAsSource(IntPtr pointer, TLink value) => (pointer +
                ↪  RightAsSourceOffset).SetValue(value);
83              [MethodImpl(MethodImplOptions.AggressiveInlining)]
84              public static void SetSizeAsSource(IntPtr pointer, TLink value) => (pointer +
                ↪  SizeAsSourceOffset).SetValue(value);
85              [MethodImpl(MethodImplOptions.AggressiveInlining)]
86              public static void SetLeftAsTarget(IntPtr pointer, TLink value) => (pointer +
                ↪  LeftAsTargetOffset).SetValue(value);
87              [MethodImpl(MethodImplOptions.AggressiveInlining)]
88              public static void SetRightAsTarget(IntPtr pointer, TLink value) => (pointer +
                ↪  RightAsTargetOffset).SetValue(value);
89              [MethodImpl(MethodImplOptions.AggressiveInlining)]
90              public static void SetSizeAsTarget(IntPtr pointer, TLink value) => (pointer +
                ↪  SizeAsTargetOffset).SetValue(value);
91          }
92
93          private struct LinksHeader
94          {
95              public static readonly int AllocatedLinksOffset =
                ↪  Marshal.OffsetOf(typeof(LinksHeader), nameof(AllocatedLinks)).ToInt32();
96              public static readonly int ReservedLinksOffset =
                ↪  Marshal.OffsetOf(typeof(LinksHeader), nameof(ReservedLinks)).ToInt32();
97              public static readonly int FreeLinksOffset = Marshal.OffsetOf(typeof(LinksHeader),
                ↪  nameof(FreeLinks)).ToInt32();
98              public static readonly int FirstFreeLinkOffset =
                ↪  Marshal.OffsetOf(typeof(LinksHeader), nameof(FirstFreeLink)).ToInt32();
99              public static readonly int FirstAsSourceOffset =
                ↪  Marshal.OffsetOf(typeof(LinksHeader), nameof(FirstAsSource)).ToInt32();
```

```
100             public static readonly int FirstAsTargetOffset =
                ↪ Marshal.OffsetOf(typeof(LinksHeader), nameof(FirstAsTarget)).ToInt32();
101             public static readonly int LastFreeLinkOffset =
                ↪ Marshal.OffsetOf(typeof(LinksHeader), nameof(LastFreeLink)).ToInt32();
102
103             public TLink AllocatedLinks;
104             public TLink ReservedLinks;
105             public TLink FreeLinks;
106             public TLink FirstFreeLink;
107             public TLink FirstAsSource;
108             public TLink FirstAsTarget;
109             public TLink LastFreeLink;
110             public TLink Reserved8;
111
112             [MethodImpl(MethodImplOptions.AggressiveInlining)]
113             public static TLink GetAllocatedLinks(IntPtr pointer) => (pointer +
                ↪ AllocatedLinksOffset).GetValue<TLink>();
114             [MethodImpl(MethodImplOptions.AggressiveInlining)]
115             public static TLink GetReservedLinks(IntPtr pointer) => (pointer +
                ↪ ReservedLinksOffset).GetValue<TLink>();
116             [MethodImpl(MethodImplOptions.AggressiveInlining)]
117             public static TLink GetFreeLinks(IntPtr pointer) => (pointer +
                ↪ FreeLinksOffset).GetValue<TLink>();
118             [MethodImpl(MethodImplOptions.AggressiveInlining)]
119             public static TLink GetFirstFreeLink(IntPtr pointer) => (pointer +
                ↪ FirstFreeLinkOffset).GetValue<TLink>();
120             [MethodImpl(MethodImplOptions.AggressiveInlining)]
121             public static TLink GetFirstAsSource(IntPtr pointer) => (pointer +
                ↪ FirstAsSourceOffset).GetValue<TLink>();
122             [MethodImpl(MethodImplOptions.AggressiveInlining)]
123             public static TLink GetFirstAsTarget(IntPtr pointer) => (pointer +
                ↪ FirstAsTargetOffset).GetValue<TLink>();
124             [MethodImpl(MethodImplOptions.AggressiveInlining)]
125             public static TLink GetLastFreeLink(IntPtr pointer) => (pointer +
                ↪ LastFreeLinkOffset).GetValue<TLink>();
126
127             [MethodImpl(MethodImplOptions.AggressiveInlining)]
128             public static IntPtr GetFirstAsSourcePointer(IntPtr pointer) => pointer +
                ↪ FirstAsSourceOffset;
129             [MethodImpl(MethodImplOptions.AggressiveInlining)]
130             public static IntPtr GetFirstAsTargetPointer(IntPtr pointer) => pointer +
                ↪ FirstAsTargetOffset;
131
132             [MethodImpl(MethodImplOptions.AggressiveInlining)]
133             public static void SetAllocatedLinks(IntPtr pointer, TLink value) => (pointer +
                ↪ AllocatedLinksOffset).SetValue(value);
134             [MethodImpl(MethodImplOptions.AggressiveInlining)]
135             public static void SetReservedLinks(IntPtr pointer, TLink value) => (pointer +
                ↪ ReservedLinksOffset).SetValue(value);
136             [MethodImpl(MethodImplOptions.AggressiveInlining)]
137             public static void SetFreeLinks(IntPtr pointer, TLink value) => (pointer +
                ↪ FreeLinksOffset).SetValue(value);
138             [MethodImpl(MethodImplOptions.AggressiveInlining)]
139             public static void SetFirstFreeLink(IntPtr pointer, TLink value) => (pointer +
                ↪ FirstFreeLinkOffset).SetValue(value);
140             [MethodImpl(MethodImplOptions.AggressiveInlining)]
141             public static void SetFirstAsSource(IntPtr pointer, TLink value) => (pointer +
                ↪ FirstAsSourceOffset).SetValue(value);
142             [MethodImpl(MethodImplOptions.AggressiveInlining)]
143             public static void SetFirstAsTarget(IntPtr pointer, TLink value) => (pointer +
                ↪ FirstAsTargetOffset).SetValue(value);
144             [MethodImpl(MethodImplOptions.AggressiveInlining)]
145             public static void SetLastFreeLink(IntPtr pointer, TLink value) => (pointer +
                ↪ LastFreeLinkOffset).SetValue(value);
146         }
147
148         private readonly long _memoryReservationStep;
149
150         private readonly IResizableDirectMemory _memory;
151         private IntPtr _header;
152         private IntPtr _links;
153
154         private LinksTargetsTreeMethods _targetsTreeMethods;
155         private LinksSourcesTreeMethods _sourcesTreeMethods;
156
157         // 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
```

```
158         private UnusedLinksListMethods _unusedLinksListMethods;
159
160         /// <summary>
161         /// Возвращает общее число связей находящихся в хранилище.
162         /// </summary>
163         private TLink Total => Subtract(LinksHeader.GetAllocatedLinks(_header),
            ↪ LinksHeader.GetFreeLinks(_header));
164
165         public LinksCombinedConstants<TLink, TLink, int> Constants { get; }
166
167         public ResizableDirectMemoryLinks(string address)
168             : this(address, DefaultLinksSizeStep)
169         {
170         }
171
172         /// <summary>
173         /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
            ↪ минимальным шагом расширения базы данных.
174         /// </summary>
175         /// <param name="address">Полный пусть к файлу базы данных.</param>
176         /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
            ↪ байтах.</param>
177         public ResizableDirectMemoryLinks(string address, long memoryReservationStep)
178             : this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
                ↪ memoryReservationStep)
179         {
180         }
181
182         public ResizableDirectMemoryLinks(IResizableDirectMemory memory)
183             : this(memory, DefaultLinksSizeStep)
184         {
185         }
186
187         public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
            ↪ memoryReservationStep)
188         {
189             Constants = Default<LinksCombinedConstants<TLink, TLink, int>>.Instance;
190             _memory = memory;
191             _memoryReservationStep = memoryReservationStep;
192             if (memory.ReservedCapacity < memoryReservationStep)
193             {
194                 memory.ReservedCapacity = memoryReservationStep;
195             }
196             SetPointers(_memory);
197             // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
198             _memory.UsedCapacity = ((long)(Integer<TLink>)LinksHeader.GetAllocatedLinks(_header)
                ↪ * LinkSizeInBytes) + LinkHeaderSizeInBytes;
199             // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
200             LinksHeader.SetReservedLinks(_header, (Integer<TLink>)((_memory.ReservedCapacity -
                ↪ LinkHeaderSizeInBytes) / LinkSizeInBytes));
201         }
202
203         [MethodImpl(MethodImplOptions.AggressiveInlining)]
204         public TLink Count(IList<TLink> restrictions)
205         {
206             // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
207             if (restrictions.Count == 0)
208             {
209                 return Total;
210             }
211             if (restrictions.Count == 1)
212             {
213                 var index = restrictions[Constants.IndexPart];
214                 if (_equalityComparer.Equals(index, Constants.Any))
215                 {
216                     return Total;
217                 }
218                 return Exists(index) ? Integer<TLink>.One : Integer<TLink>.Zero;
219             }
220             if (restrictions.Count == 2)
221             {
222                 var index = restrictions[Constants.IndexPart];
223                 var value = restrictions[1];
224                 if (_equalityComparer.Equals(index, Constants.Any))
225                 {
226                     if (_equalityComparer.Equals(value, Constants.Any))
227                     {
228                         return Total; // Any - как отсутствие ограничения
```

```
229                     }
230                     return Add(_sourcesTreeMethods.CountUsages(value),
                        ↪  _targetsTreeMethods.CountUsages(value));
231                 }
232                 else
233                 {
234                     if (!Exists(index))
235                     {
236                         return Integer<TLink>.Zero;
237                     }
238                     if (_equalityComparer.Equals(value, Constants.Any))
239                     {
240                         return Integer<TLink>.One;
241                     }
242                     var storedLinkValue = GetLinkUnsafe(index);
243                     if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
244                         _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
245                     {
246                         return Integer<TLink>.One;
247                     }
248                     return Integer<TLink>.Zero;
249                 }
250             }
251             if (restrictions.Count == 3)
252             {
253                 var index = restrictions[Constants.IndexPart];
254                 var source = restrictions[Constants.SourcePart];
255                 var target = restrictions[Constants.TargetPart];
256 
257                 if (_equalityComparer.Equals(index, Constants.Any))
258                 {
259                     if (_equalityComparer.Equals(source, Constants.Any) &&
                        ↪  _equalityComparer.Equals(target, Constants.Any))
260                     {
261                         return Total;
262                     }
263                     else if (_equalityComparer.Equals(source, Constants.Any))
264                     {
265                         return _targetsTreeMethods.CountUsages(target);
266                     }
267                     else if (_equalityComparer.Equals(target, Constants.Any))
268                     {
269                         return _sourcesTreeMethods.CountUsages(source);
270                     }
271                     else //if(source != Any && target != Any)
272                     {
273                         // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
274                         var link = _sourcesTreeMethods.Search(source, target);
275                         return _equalityComparer.Equals(link, Constants.Null) ?
                            ↪  Integer<TLink>.Zero : Integer<TLink>.One;
276                     }
277                 }
278                 else
279                 {
280                     if (!Exists(index))
281                     {
282                         return Integer<TLink>.Zero;
283                     }
284                     if (_equalityComparer.Equals(source, Constants.Any) &&
                        ↪  _equalityComparer.Equals(target, Constants.Any))
285                     {
286                         return Integer<TLink>.One;
287                     }
288                     var storedLinkValue = GetLinkUnsafe(index);
289                     if (!_equalityComparer.Equals(source, Constants.Any) &&
                        ↪  !_equalityComparer.Equals(target, Constants.Any))
290                     {
291                         if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source) &&
292                             _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
293                         {
294                             return Integer<TLink>.One;
295                         }
296                         return Integer<TLink>.Zero;
297                     }
298                     var value = default(TLink);
299                     if (_equalityComparer.Equals(source, Constants.Any))
300                     {
301                         value = target;
```

```
302                     }
303                     if (_equalityComparer.Equals(target, Constants.Any))
304                     {
305                         value = source;
306                     }
307                     if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
308                         _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
309                     {
310                         return Integer<TLink>.One;
311                     }
312                     return Integer<TLink>.Zero;
313                 }
314             }
315             throw new NotSupportedException("Другие размеры и способы ограничений не
                ↪  поддерживаются.");
316         }
317 
318         [MethodImpl(MethodImplOptions.AggressiveInlining)]
319         public TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
320         {
321             if (restrictions.Count == 0)
322             {
323                 for (TLink link = Integer<TLink>.One; _comparer.Compare(link,
                    ↪  (Integer<TLink>)LinksHeader.GetAllocatedLinks(_header)) <= 0; link =
                    ↪  Increment(link))
324                 {
325                     if (Exists(link) && _equalityComparer.Equals(handler(GetLinkStruct(link)),
                        ↪  Constants.Break))
326                     {
327                         return Constants.Break;
328                     }
329                 }
330 
331                 return Constants.Continue;
332             }
333             if (restrictions.Count == 1)
334             {
335                 var index = restrictions[Constants.IndexPart];
336                 if (_equalityComparer.Equals(index, Constants.Any))
337                 {
338                     return Each(handler, ArrayPool<TLink>.Empty);
339                 }
340                 if (!Exists(index))
341                 {
342                     return Constants.Continue;
343                 }
344                 return handler(GetLinkStruct(index));
345             }
346             if (restrictions.Count == 2)
347             {
348                 var index = restrictions[Constants.IndexPart];
349                 var value = restrictions[1];
350                 if (_equalityComparer.Equals(index, Constants.Any))
351                 {
352                     if (_equalityComparer.Equals(value, Constants.Any))
353                     {
354                         return Each(handler, ArrayPool<TLink>.Empty);
355                     }
356                     if (_equalityComparer.Equals(Each(handler, new[] { index, value,
                        ↪  Constants.Any }), Constants.Break))
357                     {
358                         return Constants.Break;
359                     }
360                     return Each(handler, new[] { index, Constants.Any, value });
361                 }
362                 else
363                 {
364                     if (!Exists(index))
365                     {
366                         return Constants.Continue;
367                     }
368                     if (_equalityComparer.Equals(value, Constants.Any))
369                     {
370                         return handler(GetLinkStruct(index));
371                     }
372                     var storedLinkValue = GetLinkUnsafe(index);
373                     if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
374                         _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
```

```
375                     {
376                         return handler(GetLinkStruct(index));
377                     }
378                     return Constants.Continue;
379                 }
380             }
381             if (restrictions.Count == 3)
382             {
383                 var index = restrictions[Constants.IndexPart];
384                 var source = restrictions[Constants.SourcePart];
385                 var target = restrictions[Constants.TargetPart];
386                 if (_equalityComparer.Equals(index, Constants.Any))
387                 {
388                     if (_equalityComparer.Equals(source, Constants.Any) &&
                        ↪ _equalityComparer.Equals(target, Constants.Any))
389                     {
390                         return Each(handler, ArrayPool<TLink>.Empty);
391                     }
392                     else if (_equalityComparer.Equals(source, Constants.Any))
393                     {
394                         return _targetsTreeMethods.EachUsage(target, handler);
395                     }
396                     else if (_equalityComparer.Equals(target, Constants.Any))
397                     {
398                         return _sourcesTreeMethods.EachUsage(source, handler);
399                     }
400                     else //if(source != Any && target != Any)
401                     {
402                         var link = _sourcesTreeMethods.Search(source, target);
403                         return _equalityComparer.Equals(link, Constants.Null) ?
                            ↪ Constants.Continue : handler(GetLinkStruct(link));
404                     }
405                 }
406                 else
407                 {
408                     if (!Exists(index))
409                     {
410                         return Constants.Continue;
411                     }
412                     if (_equalityComparer.Equals(source, Constants.Any) &&
                        ↪ _equalityComparer.Equals(target, Constants.Any))
413                     {
414                         return handler(GetLinkStruct(index));
415                     }
416                     var storedLinkValue = GetLinkUnsafe(index);
417                     if (!_equalityComparer.Equals(source, Constants.Any) &&
                        ↪ !_equalityComparer.Equals(target, Constants.Any))
418                     {
419                         if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source) &&
420                             _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
421                         {
422                             return handler(GetLinkStruct(index));
423                         }
424                         return Constants.Continue;
425                     }
426                     var value = default(TLink);
427                     if (_equalityComparer.Equals(source, Constants.Any))
428                     {
429                         value = target;
430                     }
431                     if (_equalityComparer.Equals(target, Constants.Any))
432                     {
433                         value = source;
434                     }
435                     if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
436                         _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
437                     {
438                         return handler(GetLinkStruct(index));
439                     }
440                     return Constants.Continue;
441                 }
442             }
443             throw new NotSupportedException("Другие размеры и способы ограничений не
                ↪ поддерживаются.");
444         }
445
446         /// <remarks>
```

```
447         /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
            ↪ в другом месте (но не в менеджере памяти, а в логике Links)
448         /// </remarks>
449         [MethodImpl(MethodImplOptions.AggressiveInlining)]
450         public TLink Update(IList<TLink> values)
451         {
452             var linkIndex = values[Constants.IndexPart];
453             var link = GetLinkUnsafe(linkIndex);
454             // Будет корректно работать только в том случае, если пространство выделенной связи
                ↪ предварительно заполнено нулями
455             if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
456             {
457                 _sourcesTreeMethods.Detach(LinksHeader.GetFirstAsSourcePointer(_header),
                    ↪ linkIndex);
458             }
459             if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
460             {
461                 _targetsTreeMethods.Detach(LinksHeader.GetFirstAsTargetPointer(_header),
                    ↪ linkIndex);
462             }
463             Link.SetSource(link, values[Constants.SourcePart]);
464             Link.SetTarget(link, values[Constants.TargetPart]);
465             if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
466             {
467                 _sourcesTreeMethods.Attach(LinksHeader.GetFirstAsSourcePointer(_header),
                    ↪ linkIndex);
468             }
469             if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
470             {
471                 _targetsTreeMethods.Attach(LinksHeader.GetFirstAsTargetPointer(_header),
                    ↪ linkIndex);
472             }
473             return linkIndex;
474         }
475
476         [MethodImpl(MethodImplOptions.AggressiveInlining)]
477         public Link<TLink> GetLinkStruct(TLink linkIndex)
478         {
479             var link = GetLinkUnsafe(linkIndex);
480             return new Link<TLink>(linkIndex, Link.GetSource(link), Link.GetTarget(link));
481         }
482
483         [MethodImpl(MethodImplOptions.AggressiveInlining)]
484         private IntPtr GetLinkUnsafe(TLink linkIndex) => _links.GetElement(LinkSizeInBytes,
            ↪ linkIndex);
485
486         /// <remarks>
487         /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
            ↪ пространство
488         /// </remarks>
489         public TLink Create()
490         {
491             var freeLink = LinksHeader.GetFirstFreeLink(_header);
492             if (!_equalityComparer.Equals(freeLink, Constants.Null))
493             {
494                 _unusedLinksListMethods.Detach(freeLink);
495             }
496             else
497             {
498                 if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
                    ↪ Constants.MaxPossibleIndex) > 0)
499                 {
500                     throw new
                        ↪ LinksLimitReachedException((Integer<TLink>)Constants.MaxPossibleIndex);
501                 }
502                 if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
                    ↪ Decrement(LinksHeader.GetReservedLinks(_header))) >= 0)
503                 {
504                     _memory.ReservedCapacity += _memoryReservationStep;
505                     SetPointers(_memory);
506                     LinksHeader.SetReservedLinks(_header,
                        ↪ (Integer<TLink>)(_memory.ReservedCapacity / LinkSizeInBytes));
507                 }
508                 LinksHeader.SetAllocatedLinks(_header,
                    ↪ Increment(LinksHeader.GetAllocatedLinks(_header)));
509                 _memory.UsedCapacity += LinkSizeInBytes;
510                 freeLink = LinksHeader.GetAllocatedLinks(_header);
```

```
511             }
512             return freeLink;
513         }
514 
515         public void Delete(TLink link)
516         {
517             if (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) < 0)
518             {
519                 _unusedLinksListMethods.AttachAsFirst(link);
520             }
521             else if (_equalityComparer.Equals(link, LinksHeader.GetAllocatedLinks(_header)))
522             {
523                 LinksHeader.SetAllocatedLinks(_header,
                    ↪  Decrement(LinksHeader.GetAllocatedLinks(_header)));
524                 _memory.UsedCapacity -= LinkSizeInBytes;
525                 // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
                    ↪  пока не дойдём до первой существующей связи
526                 // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
527                 while ((_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
                    ↪  Integer<TLink>.Zero) > 0) &&
                    ↪  IsUnusedLink(LinksHeader.GetAllocatedLinks(_header)))
528                 {
529                     _unusedLinksListMethods.Detach(LinksHeader.GetAllocatedLinks(_header));
530                     LinksHeader.SetAllocatedLinks(_header,
                        ↪  Decrement(LinksHeader.GetAllocatedLinks(_header)));
531                     _memory.UsedCapacity -= LinkSizeInBytes;
532                 }
533             }
534         }
535 
536         /// <remarks>
537         /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
            ↪  адрес реально поменялся
538         ///
539         /// Указатель this.links может быть в том же месте,
540         /// так как 0-я связь не используется и имеет такой же размер как Header,
541         /// поэтому header размещается в том же месте, что и 0-я связь
542         /// </remarks>
543         private void SetPointers(IDirectMemory memory)
544         {
545             if (memory == null)
546             {
547                 _links = IntPtr.Zero;
548                 _header = _links;
549                 _unusedLinksListMethods = null;
550                 _targetsTreeMethods = null;
551                 _unusedLinksListMethods = null;
552             }
553             else
554             {
555                 _links = memory.Pointer;
556                 _header = _links;
557                 _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
558                 _targetsTreeMethods = new LinksTargetsTreeMethods(this);
559                 _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
560             }
561         }
562 
563         [MethodImpl(MethodImplOptions.AggressiveInlining)]
564         private bool Exists(TLink link)
565             => (_comparer.Compare(link, Constants.MinPossibleIndex) >= 0)
566             && (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) <= 0)
567             && !IsUnusedLink(link);
568 
569         [MethodImpl(MethodImplOptions.AggressiveInlining)]
570         private bool IsUnusedLink(TLink link)
571             => _equalityComparer.Equals(LinksHeader.GetFirstFreeLink(_header), link)
572             || (_equalityComparer.Equals(Link.GetSizeAsSource(GetLinkUnsafe(link)),
                ↪  Constants.Null)
573             && !_equalityComparer.Equals(Link.GetSource(GetLinkUnsafe(link)), Constants.Null));
574 
575         #region DisposableBase
576 
577         protected override bool AllowMultipleDisposeCalls => true;
578 
579         protected override void Dispose(bool manual, bool wasDisposed)
580         {
581             if (!wasDisposed)
582             {
```

```
                SetPointers(null);
                _memory.DisposeIfPossible();
            }
        }

        #endregion
    }
}
```

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

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

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;

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    partial class ResizableDirectMemoryLinks<TLink>
    {
```

```
14          private abstract class LinksTreeMethodsBase :
            ↪ SizedAndThreadedAVLBalancedTreeMethods<TLink>
15          {
16              private readonly ResizableDirectMemoryLinks<TLink> _memory;
17              private readonly LinksCombinedConstants<TLink, TLink, int> _constants;
18              protected readonly IntPtr Links;
19              protected readonly IntPtr Header;
20
21              protected LinksTreeMethodsBase(ResizableDirectMemoryLinks<TLink> memory)
22              {
23                  Links = memory._links;
24                  Header = memory._header;
25                  _memory = memory;
26                  _constants = memory.Constants;
27              }
28
29              [MethodImpl(MethodImplOptions.AggressiveInlining)]
30              protected abstract TLink GetTreeRoot();
31
32              [MethodImpl(MethodImplOptions.AggressiveInlining)]
33              protected abstract TLink GetBasePartValue(TLink link);
34
35              public TLink this[TLink index]
36              {
37                  get
38                  {
39                      var root = GetTreeRoot();
40                      if (GreaterOrEqualThan(index, GetSize(root)))
41                      {
42                          return GetZero();
43                      }
44                      while (!EqualToZero(root))
45                      {
46                          var left = GetLeftOrDefault(root);
47                          var leftSize = GetSizeOrZero(left);
48                          if (LessThan(index, leftSize))
49                          {
50                              root = left;
51                              continue;
52                          }
53                          if (IsEquals(index, leftSize))
54                          {
55                              return root;
56                          }
57                          root = GetRightOrDefault(root);
58                          index = Subtract(index, Increment(leftSize));
59                      }
60                      return GetZero(); // TODO: Impossible situation exception (only if tree
                        ↪ structure broken)
61                  }
62              }
63
64              // TODO: Return indices range instead of references count
65              public TLink CountUsages(TLink link)
66              {
67                  var root = GetTreeRoot();
68                  var total = GetSize(root);
69                  var totalRightIgnore = GetZero();
70                  while (!EqualToZero(root))
71                  {
72                      var @base = GetBasePartValue(root);
73                      if (LessOrEqualThan(@base, link))
74                      {
75                          root = GetRightOrDefault(root);
76                      }
77                      else
78                      {
79                          totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
80                          root = GetLeftOrDefault(root);
81                      }
82                  }
83                  root = GetTreeRoot();
84                  var totalLeftIgnore = GetZero();
85                  while (!EqualToZero(root))
86                  {
87                      var @base = GetBasePartValue(root);
88                      if (GreaterOrEqualThan(@base, link))
89                      {
90                          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;

```

```
166             protected override TLink GetLeftValue(TLink node) =>
                ↪  (Links.GetElement(LinkSizeInBytes, node) +
                ↪  Link.LeftAsSourceOffset).GetValue<TLink>();
167
168             protected override TLink GetRightValue(TLink node) =>
                ↪  (Links.GetElement(LinkSizeInBytes, node) +
                ↪  Link.RightAsSourceOffset).GetValue<TLink>();
169
170             protected override TLink GetSize(TLink node)
171             {
172                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsSourceOffset).GetValue<TLink>();
173                 return Bit.PartialRead(previousValue, 5, -5);
174             }
175
176             protected override void SetLeft(TLink node, TLink left) =>
                ↪  (Links.GetElement(LinkSizeInBytes, node) +
                ↪  Link.LeftAsSourceOffset).SetValue(left);
177
178             protected override void SetRight(TLink node, TLink right) =>
                ↪  (Links.GetElement(LinkSizeInBytes, node) +
                ↪  Link.RightAsSourceOffset).SetValue(right);
179
180             protected override void SetSize(TLink node, TLink size)
181             {
182                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsSourceOffset).GetValue<TLink>();
183                 (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsSourceOffset).SetValue(Bit.PartialWrite(previousValue, size, 5,
                    ↪  -5));
184             }
185
186             protected override bool GetLeftIsChild(TLink node)
187             {
188                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsSourceOffset).GetValue<TLink>();
189                 return (Integer<TLink>)Bit.PartialRead(previousValue, 4, 1);
190             }
191
192             protected override void SetLeftIsChild(TLink node, bool value)
193             {
194                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsSourceOffset).GetValue<TLink>();
195                 var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 4,
                    ↪  1);
196                 (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsSourceOffset).SetValue(modified);
197             }
198
199             protected override bool GetRightIsChild(TLink node)
200             {
201                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsSourceOffset).GetValue<TLink>();
202                 return (Integer<TLink>)Bit.PartialRead(previousValue, 3, 1);
203             }
204
205             protected override void SetRightIsChild(TLink node, bool value)
206             {
207                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsSourceOffset).GetValue<TLink>();
208                 var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 3,
                    ↪  1);
209                 (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsSourceOffset).SetValue(modified);
210             }
211
212             protected override sbyte GetBalance(TLink node)
213             {
214                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsSourceOffset).GetValue<TLink>();
215                 var value = (ulong)(Integer<TLink>)Bit.PartialRead(previousValue, 0, 3);
216                 var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
                    ↪  124 : value & 3);
217                 return unpackedValue;
218             }
219
220             protected override void SetBalance(TLink node, sbyte value)
```

```
221             {
222                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
223                 var packagedValue = (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) | value &
                    ↪ 3);
224                 var modified = Bit.PartialWrite(previousValue, packagedValue, 0, 3);
225                 (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsSourceOffset).SetValue(modified);
226             }
227
228             protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
229             {
230                 var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
                    ↪ Link.SourceOffset).GetValue<TLink>();
231                 var secondSource = (Links.GetElement(LinkSizeInBytes, second) +
                    ↪ Link.SourceOffset).GetValue<TLink>();
232                 return LessThan(firstSource, secondSource) ||
233                        (IsEquals(firstSource, secondSource) &&
                           ↪ LessThan((Links.GetElement(LinkSizeInBytes, first) +
                           ↪ Link.TargetOffset).GetValue<TLink>(),
                           ↪ (Links.GetElement(LinkSizeInBytes, second) +
                           ↪ Link.TargetOffset).GetValue<TLink>()));
234             }
235
236             protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
237             {
238                 var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
                    ↪ Link.SourceOffset).GetValue<TLink>();
239                 var secondSource = (Links.GetElement(LinkSizeInBytes, second) +
                    ↪ Link.SourceOffset).GetValue<TLink>();
240                 return GreaterThan(firstSource, secondSource) ||
241                        (IsEquals(firstSource, secondSource) &&
                           ↪ GreaterThan((Links.GetElement(LinkSizeInBytes, first) +
                           ↪ Link.TargetOffset).GetValue<TLink>(),
                           ↪ (Links.GetElement(LinkSizeInBytes, second) +
                           ↪ Link.TargetOffset).GetValue<TLink>()));
242             }
243
244             protected override TLink GetTreeRoot() => (Header +
                ↪ LinksHeader.FirstAsSourceOffset).GetValue<TLink>();
245
246             protected override TLink GetBasePartValue(TLink link) =>
                ↪ (Links.GetElement(LinkSizeInBytes, link) + Link.SourceOffset).GetValue<TLink>();
247
248             /// <summary>
249             /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
                ↪ (концом)
250             /// по дереву (индексу) связей, отсортированному по Source, а затем по Target.
251             /// </summary>
252             /// <param name="source">Индекс связи, которая является началом на искомой
                ↪ связи.</param>
253             /// <param name="target">Индекс связи, которая является концом на искомой
                ↪ связи.</param>
254             /// <returns>Индекс искомой связи.</returns>
255             public TLink Search(TLink source, TLink target)
256             {
257                 var root = GetTreeRoot();
258                 while (!EqualToZero(root))
259                 {
260                     var rootSource = (Links.GetElement(LinkSizeInBytes, root) +
                        ↪ Link.SourceOffset).GetValue<TLink>();
261                     var rootTarget = (Links.GetElement(LinkSizeInBytes, root) +
                        ↪ Link.TargetOffset).GetValue<TLink>();
262                     if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                        ↪ node.Key < root.Key
263                     {
264                         root = GetLeftOrDefault(root);
265                     }
266                     else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget))
                        ↪ // node.Key > root.Key
267                     {
268                         root = GetRightOrDefault(root);
269                     }
270                     else // node.Key == root.Key
271                     {
272                         return root;
```

```
273                     }
274                 }
275                 return GetZero();
276             }
277 
278             [MethodImpl(MethodImplOptions.AggressiveInlining)]
279             private bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget, TLink
                ↪  secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
                ↪  (IsEquals(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
280 
281             [MethodImpl(MethodImplOptions.AggressiveInlining)]
282             private bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget, TLink
                ↪  secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
                ↪  (IsEquals(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
283         }
284 
285         private class LinksTargetsTreeMethods : LinksTreeMethodsBase
286         {
287             public LinksTargetsTreeMethods(ResizableDirectMemoryLinks<TLink> memory)
288                 : base(memory)
289             {
290             }
291 
292             protected override IntPtr GetLeftPointer(TLink node) =>
                ↪  Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsTargetOffset;
293 
294             protected override IntPtr GetRightPointer(TLink node) =>
                ↪  Links.GetElement(LinkSizeInBytes, node) + Link.RightAsTargetOffset;
295 
296             protected override TLink GetLeftValue(TLink node) =>
                ↪  (Links.GetElement(LinkSizeInBytes, node) +
                ↪  Link.LeftAsTargetOffset).GetValue<TLink>();
297 
298             protected override TLink GetRightValue(TLink node) =>
                ↪  (Links.GetElement(LinkSizeInBytes, node) +
                ↪  Link.RightAsTargetOffset).GetValue<TLink>();
299 
300             protected override TLink GetSize(TLink node)
301             {
302                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsTargetOffset).GetValue<TLink>();
303                 return Bit.PartialRead(previousValue, 5, -5);
304             }
305 
306             protected override void SetLeft(TLink node, TLink left) =>
                ↪  (Links.GetElement(LinkSizeInBytes, node) +
                ↪  Link.LeftAsTargetOffset).SetValue(left);
307 
308             protected override void SetRight(TLink node, TLink right) =>
                ↪  (Links.GetElement(LinkSizeInBytes, node) +
                ↪  Link.RightAsTargetOffset).SetValue(right);
309 
310             protected override void SetSize(TLink node, TLink size)
311             {
312                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsTargetOffset).GetValue<TLink>();
313                 (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsTargetOffset).SetValue(Bit.PartialWrite(previousValue, size, 5,
                    ↪  -5));
314             }
315 
316             protected override bool GetLeftIsChild(TLink node)
317             {
318                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsTargetOffset).GetValue<TLink>();
319                 return (Integer<TLink>)Bit.PartialRead(previousValue, 4, 1);
320             }
321 
322             protected override void SetLeftIsChild(TLink node, bool value)
323             {
324                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsTargetOffset).GetValue<TLink>();
325                 var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 4,
                    ↪  1);
326                 (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsTargetOffset).SetValue(modified);
327             }
```

```
328
329             protected override bool GetRightIsChild(TLink node)
330             {
331                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
332                 return (Integer<TLink>)Bit.PartialRead(previousValue, 3, 1);
333             }
334
335             protected override void SetRightIsChild(TLink node, bool value)
336             {
337                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
338                 var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 3,
                    ↪ 1);
339                 (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsTargetOffset).SetValue(modified);
340             }
341
342             protected override sbyte GetBalance(TLink node)
343             {
344                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
345                 var value = (ulong)(Integer<TLink>)Bit.PartialRead(previousValue, 0, 3);
346                 var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
                    ↪ 124 : value & 3);
347                 return unpackedValue;
348             }
349
350             protected override void SetBalance(TLink node, sbyte value)
351             {
352                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
353                 var packagedValue = (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) | value &
                    ↪ 3);
354                 var modified = Bit.PartialWrite(previousValue, packagedValue, 0, 3);
355                 (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsTargetOffset).SetValue(modified);
356             }
357
358             protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
359             {
360                 var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
                    ↪ Link.TargetOffset).GetValue<TLink>();
361                 var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
                    ↪ Link.TargetOffset).GetValue<TLink>();
362                 return LessThan(firstTarget, secondTarget) ||
363                        (IsEquals(firstTarget, secondTarget) &&
                           ↪ LessThan((Links.GetElement(LinkSizeInBytes, first) +
                           ↪ Link.SourceOffset).GetValue<TLink>(),
                           ↪ (Links.GetElement(LinkSizeInBytes, second) +
                           ↪ Link.SourceOffset).GetValue<TLink>()));
364             }
365
366             protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
367             {
368                 var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
                    ↪ Link.TargetOffset).GetValue<TLink>();
369                 var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
                    ↪ Link.TargetOffset).GetValue<TLink>();
370                 return GreaterThan(firstTarget, secondTarget) ||
371                        (IsEquals(firstTarget, secondTarget) &&
                           ↪ GreaterThan((Links.GetElement(LinkSizeInBytes, first) +
                           ↪ Link.SourceOffset).GetValue<TLink>(),
                           ↪ (Links.GetElement(LinkSizeInBytes, second) +
                           ↪ Link.SourceOffset).GetValue<TLink>()));
372             }
373
374             protected override TLink GetTreeRoot() => (Header +
                ↪ LinksHeader.FirstAsTargetOffset).GetValue<TLink>();
375
376             protected override TLink GetBasePartValue(TLink link) =>
                ↪ (Links.GetElement(LinkSizeInBytes, link) + Link.TargetOffset).GetValue<TLink>();
377         }
378     }
379 }
```

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

```
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;

//#define ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION

#pragma warning disable 0649
#pragma warning disable 169

// ReSharper disable BuiltInTypeReferenceStyle

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

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

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

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

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

        private readonly long _memoryReservationStep;

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

        private LinksTargetsTreeMethods _targetsTreeMethods;
        private LinksSourcesTreeMethods _sourcesTreeMethods;

        // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
        ↪  нужно использовать не список а дерево, так как так можно быстрее проверить на
        ↪  наличие связи внутри
        private UnusedLinksListMethods _unusedLinksListMethods;

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

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

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

```

```
79          /// <summary>
80          /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
            ↪ минимальным шагом расширения базы данных.
81          /// </summary>
82          /// <param name="address">Полный пусть к файлу базы данных.</param>
83          /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
            ↪ байтах.</param>
84          public UInt64ResizableDirectMemoryLinks(string address, long memoryReservationStep) :
            ↪ this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
            ↪ memoryReservationStep) { }
85
86          public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
            ↪ DefaultLinksSizeStep) { }
87
88          public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
            ↪ memoryReservationStep)
89          {
90              Constants = Default<LinksCombinedConstants<id, id, int>>.Instance;
91              _memory = memory;
92              _memoryReservationStep = memoryReservationStep;
93              if (memory.ReservedCapacity < memoryReservationStep)
94              {
95                  memory.ReservedCapacity = memoryReservationStep;
96              }
97              SetPointers(_memory);
98              // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
99              _memory.UsedCapacity = ((long)_header->AllocatedLinks * sizeof(Link)) +
                ↪ sizeof(LinksHeader);
100             // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
101             _header->ReservedLinks = (id)((_memory.ReservedCapacity - sizeof(LinksHeader)) /
                ↪ sizeof(Link));
102         }
103
104         [MethodImpl(MethodImplOptions.AggressiveInlining)]
105         public id Count(IList<id> restrictions)
106         {
107             // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
108             if (restrictions.Count == 0)
109             {
110                 return Total;
111             }
112             if (restrictions.Count == 1)
113             {
114                 var index = restrictions[Constants.IndexPart];
115                 if (index == Constants.Any)
116                 {
117                     return Total;
118                 }
119                 return Exists(index) ? 1UL : 0UL;
120             }
121             if (restrictions.Count == 2)
122             {
123                 var index = restrictions[Constants.IndexPart];
124                 var value = restrictions[1];
125                 if (index == Constants.Any)
126                 {
127                     if (value == Constants.Any)
128                     {
129                         return Total; // Any - как отсутствие ограничения
130                     }
131                     return _sourcesTreeMethods.CountUsages(value)
132                          + _targetsTreeMethods.CountUsages(value);
133                 }
134                 else
135                 {
136                     if (!Exists(index))
137                     {
138                         return 0;
139                     }
140                     if (value == Constants.Any)
141                     {
142                         return 1;
143                     }
144                     var storedLinkValue = GetLinkUnsafe(index);
145                     if (storedLinkValue->Source == value ||
146                         storedLinkValue->Target == value)
147                     {
148                         return 1;
```

```
                    }
                    return 0;
                }
            }
            if (restrictions.Count == 3)
            {
                var index = restrictions[Constants.IndexPart];
                var source = restrictions[Constants.SourcePart];
                var target = restrictions[Constants.TargetPart];
                if (index == Constants.Any)
                {
                    if (source == Constants.Any && target == Constants.Any)
                    {
                        return Total;
                    }
                    else if (source == Constants.Any)
                    {
                        return _targetsTreeMethods.CountUsages(target);
                    }
                    else if (target == Constants.Any)
                    {
                        return _sourcesTreeMethods.CountUsages(source);
                    }
                    else //if(source != Any && target != Any)
                    {
                        // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
                        var link = _sourcesTreeMethods.Search(source, target);
                        return link == Constants.Null ? 0UL : 1UL;
                    }
                }
                else
                {
                    if (!Exists(index))
                    {
                        return 0;
                    }
                    if (source == Constants.Any && target == Constants.Any)
                    {
                        return 1;
                    }
                    var storedLinkValue = GetLinkUnsafe(index);
                    if (source != Constants.Any && target != Constants.Any)
                    {
                        if (storedLinkValue->Source == source &&
                            storedLinkValue->Target == target)
                        {
                            return 1;
                        }
                        return 0;
                    }
                    var value = default(id);
                    if (source == Constants.Any)
                    {
                        value = target;
                    }
                    if (target == Constants.Any)
                    {
                        value = source;
                    }
                    if (storedLinkValue->Source == value ||
                        storedLinkValue->Target == value)
                    {
                        return 1;
                    }
                    return 0;
                }
            }
            throw new NotSupportedException("Другие размеры и способы ограничений не
            ↪  поддерживаются.");
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public id Each(Func<IList<id>, id> handler, IList<id> restrictions)
        {
            if (restrictions.Count == 0)
            {
                for (id link = 1; link <= _header->AllocatedLinks; link++)
                {
                    if (Exists(link))
```

```
                {
                    if (handler(GetLinkStruct(link)) == Constants.Break)
                    {
                        return Constants.Break;
                    }
                }
            }
            return Constants.Continue;
        }
        if (restrictions.Count == 1)
        {
            var index = restrictions[Constants.IndexPart];
            if (index == Constants.Any)
            {
                return Each(handler, ArrayPool<ulong>.Empty);
            }
            if (!Exists(index))
            {
                return Constants.Continue;
            }
            return handler(GetLinkStruct(index));
        }
        if (restrictions.Count == 2)
        {
            var index = restrictions[Constants.IndexPart];
            var value = restrictions[1];
            if (index == Constants.Any)
            {
                if (value == Constants.Any)
                {
                    return Each(handler, ArrayPool<ulong>.Empty);
                }
                if (Each(handler, new[] { index, value, Constants.Any }) == Constants.Break)
                {
                    return Constants.Break;
                }
                return Each(handler, new[] { index, Constants.Any, value });
            }
            else
            {
                if (!Exists(index))
                {
                    return Constants.Continue;
                }
                if (value == Constants.Any)
                {
                    return handler(GetLinkStruct(index));
                }
                var storedLinkValue = GetLinkUnsafe(index);
                if (storedLinkValue->Source == value ||
                    storedLinkValue->Target == value)
                {
                    return handler(GetLinkStruct(index));
                }
                return Constants.Continue;
            }
        }
        if (restrictions.Count == 3)
        {
            var index = restrictions[Constants.IndexPart];
            var source = restrictions[Constants.SourcePart];
            var target = restrictions[Constants.TargetPart];
            if (index == Constants.Any)
            {
                if (source == Constants.Any && target == Constants.Any)
                {
                    return Each(handler, ArrayPool<ulong>.Empty);
                }
                else if (source == Constants.Any)
                {
                    return _targetsTreeMethods.EachReference(target, handler);
                }
                else if (target == Constants.Any)
                {
                    return _sourcesTreeMethods.EachReference(source, handler);
                }
                else //if(source != Any && target != Any)
                {
```

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

```
            }
            if (link->Target != Constants.Null)
            {
                _targetsTreeMethods.Attach(new IntPtr(&_header->FirstAsTarget), linkIndex);
            }
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
            leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
            rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
            if (leftTreeSize != rightTreeSize)
            {
                throw new Exception("One of the trees is broken.");
            }
#endif
            return linkIndex;
        }

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

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

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

        public void Delete(id link)
        {
            if (link < _header->AllocatedLinks)
            {
                _unusedLinksListMethods.AttachAsFirst(link);
            }
            else if (link == _header->AllocatedLinks)
            {
                _header->AllocatedLinks--;
                _memory.UsedCapacity -= sizeof(Link);
                // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
                ↪  пока не дойдём до первой существующей связи
                // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
                while (_header->AllocatedLinks > 0 && IsUnusedLink(_header->AllocatedLinks))
                {
                    _unusedLinksListMethods.Detach(_header->AllocatedLinks);
                    _header->AllocatedLinks--;
                    _memory.UsedCapacity -= sizeof(Link);
                }
            }
        }

        /// <remarks>
```

```
456         /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
            ↪  адрес реально поменялся
457         ///
458         /// Указатель this.links может быть в том же месте,
459         /// так как 0-я связь не используется и имеет такой же размер как Header,
460         /// поэтому header размещается в том же месте, что и 0-я связь
461         /// </remarks>
462         private void SetPointers(IResizableDirectMemory memory)
463         {
464             if (memory == null)
465             {
466                 _header = null;
467                 _links = null;
468                 _unusedLinksListMethods = null;
469                 _targetsTreeMethods = null;
470                 _unusedLinksListMethods = null;
471             }
472             else
473             {
474                 _header = (LinksHeader*)(void*)memory.Pointer;
475                 _links = (Link*)(void*)memory.Pointer;
476                 _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
477                 _targetsTreeMethods = new LinksTargetsTreeMethods(this);
478                 _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
479             }
480         }
481
482         [MethodImpl(MethodImplOptions.AggressiveInlining)]
483         private bool Exists(id link) => link >= Constants.MinPossibleIndex && link <=
            ↪  _header->AllocatedLinks && !IsUnusedLink(link);
484
485         [MethodImpl(MethodImplOptions.AggressiveInlining)]
486         private bool IsUnusedLink(id link) => _header->FirstFreeLink == link
487                                               || (_links[link].SizeAsSource == Constants.Null &&
                                                  ↪  _links[link].Source != Constants.Null);
488
489         #region Disposable
490
491         protected override bool AllowMultipleDisposeCalls => true;
492
493         protected override void Dispose(bool manual, bool wasDisposed)
494         {
495             if (!wasDisposed)
496             {
497                 SetPointers(null);
498                 _memory.DisposeIfPossible();
499             }
500         }
501
502         #endregion
503     }
504 }
```

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

```
1  using Platform.Collections.Methods.Lists;
2
3  namespace Platform.Data.Doublets.ResizableDirectMemory
4  {
5      unsafe partial class UInt64ResizableDirectMemoryLinks
6      {
7          private class UnusedLinksListMethods : CircularDoublyLinkedListMethods<ulong>
8          {
9              private readonly Link* _links;
10             private readonly LinksHeader* _header;
11
12             public UnusedLinksListMethods(Link* links, LinksHeader* header)
13             {
14                 _links = links;
15                 _header = header;
16             }
17
18             protected override ulong GetFirst() => _header->FirstFreeLink;
19
20             protected override ulong GetLast() => _header->LastFreeLink;
21
22             protected override ulong GetPrevious(ulong element) => _links[element].Source;
23
24             protected override ulong GetNext(ulong element) => _links[element].Target;
25
26             protected override ulong GetSize() => _header->FreeLinks;
```

```
27
28              protected override void SetFirst(ulong element) => _header->FirstFreeLink = element;
29
30              protected override void SetLast(ulong element) => _header->LastFreeLink = element;
31
32              protected override void SetPrevious(ulong element, ulong previous) =>
                ↪  _links[element].Source = previous;
33
34              protected override void SetNext(ulong element, ulong next) => _links[element].Target
                ↪  = next;
35
36              protected override void SetSize(ulong size) => _header->FreeLinks = size;
37          }
38      }
39  }
```

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

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

```
        // 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;
        }
```

```
141
142             protected override void PrintNodeValue(ulong node, StringBuilder sb)
143             {
144                 sb.Append(' ');
145                 sb.Append(Links[node].Source);
146                 sb.Append('-');
147                 sb.Append('>');
148                 sb.Append(Links[node].Target);
149             }
150         }
151
152         private class LinksSourcesTreeMethods : LinksTreeMethodsBase
153         {
154             public LinksSourcesTreeMethods(UInt64ResizableDirectMemoryLinks memory)
155                 : base(memory)
156             {
157             }
158
159             protected override IntPtr GetLeftPointer(ulong node) => new
                ↪  IntPtr(&Links[node].LeftAsSource);
160
161             protected override IntPtr GetRightPointer(ulong node) => new
                ↪  IntPtr(&Links[node].RightAsSource);
162
163             protected override ulong GetLeftValue(ulong node) => Links[node].LeftAsSource;
164
165             protected override ulong GetRightValue(ulong node) => Links[node].RightAsSource;
166
167             protected override ulong GetSize(ulong node)
168             {
169                 var previousValue = Links[node].SizeAsSource;
170                 //return Math.PartialRead(previousValue, 5, -5);
171                 return (previousValue & 4294967264) >> 5;
172             }
173
174             protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource
                ↪  = left;
175
176             protected override void SetRight(ulong node, ulong right) =>
                ↪  Links[node].RightAsSource = right;
177
178             protected override void SetSize(ulong node, ulong size)
179             {
180                 var previousValue = Links[node].SizeAsSource;
181                 //var modified = Math.PartialWrite(previousValue, size, 5, -5);
182                 var modified = (previousValue & 31) | ((size & 134217727) << 5);
183                 Links[node].SizeAsSource = modified;
184             }
185
186             protected override bool GetLeftIsChild(ulong node)
187             {
188                 var previousValue = Links[node].SizeAsSource;
189                 //return (Integer)Math.PartialRead(previousValue, 4, 1);
190                 return (previousValue & 16) >> 4 == 1UL;
191             }
192
193             protected override void SetLeftIsChild(ulong node, bool value)
194             {
195                 var previousValue = Links[node].SizeAsSource;
196                 //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 4, 1);
197                 var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
198                 Links[node].SizeAsSource = modified;
199             }
200
201             protected override bool GetRightIsChild(ulong node)
202             {
203                 var previousValue = Links[node].SizeAsSource;
204                 //return (Integer)Math.PartialRead(previousValue, 3, 1);
205                 return (previousValue & 8) >> 3 == 1UL;
206             }
207
208             protected override void SetRightIsChild(ulong node, bool value)
209             {
210                 var previousValue = Links[node].SizeAsSource;
211                 //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 3, 1);
212                 var modified = (previousValue & 4294967287) | ((value ? 1UL : 0UL) << 3);
213                 Links[node].SizeAsSource = modified;
214             }
215
```

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

```
282                 => firstSource > secondSource || (firstSource == secondSource && firstTarget >
                    ↪  secondTarget);
283
284             [MethodImpl(MethodImplOptions.AggressiveInlining)]
285             protected override void ClearNode(ulong node)
286             {
287                 Links[node].LeftAsSource = 0UL;
288                 Links[node].RightAsSource = 0UL;
289                 Links[node].SizeAsSource = 0UL;
290             }
291
292             [MethodImpl(MethodImplOptions.AggressiveInlining)]
293             protected override ulong GetZero() => 0UL;
294
295             [MethodImpl(MethodImplOptions.AggressiveInlining)]
296             protected override ulong GetOne() => 1UL;
297
298             [MethodImpl(MethodImplOptions.AggressiveInlining)]
299             protected override ulong GetTwo() => 2UL;
300
301             [MethodImpl(MethodImplOptions.AggressiveInlining)]
302             protected override bool ValueEqualToZero(IntPtr pointer) =>
                ↪  *(ulong*)pointer.ToPointer() == 0UL;
303
304             [MethodImpl(MethodImplOptions.AggressiveInlining)]
305             protected override bool EqualToZero(ulong value) => value == 0UL;
306
307             [MethodImpl(MethodImplOptions.AggressiveInlining)]
308             protected override bool IsEquals(ulong first, ulong second) => first == second;
309
310             [MethodImpl(MethodImplOptions.AggressiveInlining)]
311             protected override bool GreaterThanZero(ulong value) => value > 0UL;
312
313             [MethodImpl(MethodImplOptions.AggressiveInlining)]
314             protected override bool GreaterThan(ulong first, ulong second) => first > second;
315
316             [MethodImpl(MethodImplOptions.AggressiveInlining)]
317             protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >=
                ↪  second;
318
319             [MethodImpl(MethodImplOptions.AggressiveInlining)]
320             protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0
                ↪  is always true for ulong
321
322             [MethodImpl(MethodImplOptions.AggressiveInlining)]
323             protected override bool LessOrEqualThanZero(ulong value) => value == 0; // value is
                ↪  always >= 0 for ulong
324
325             [MethodImpl(MethodImplOptions.AggressiveInlining)]
326             protected override bool LessOrEqualThan(ulong first, ulong second) => first <=
                ↪  second;
327
328             [MethodImpl(MethodImplOptions.AggressiveInlining)]
329             protected override bool LessThanZero(ulong value) => false; // value < 0 is always
                ↪  false for ulong
330
331             [MethodImpl(MethodImplOptions.AggressiveInlining)]
332             protected override bool LessThan(ulong first, ulong second) => first < second;
333
334             [MethodImpl(MethodImplOptions.AggressiveInlining)]
335             protected override ulong Increment(ulong value) => ++value;
336
337             [MethodImpl(MethodImplOptions.AggressiveInlining)]
338             protected override ulong Decrement(ulong value) => --value;
339
340             [MethodImpl(MethodImplOptions.AggressiveInlining)]
341             protected override ulong Add(ulong first, ulong second) => first + second;
342
343             [MethodImpl(MethodImplOptions.AggressiveInlining)]
344             protected override ulong Subtract(ulong first, ulong second) => first - second;
345         }
346
347         private class LinksTargetsTreeMethods : LinksTreeMethodsBase
348         {
349             public LinksTargetsTreeMethods(UInt64ResizableDirectMemoryLinks memory)
350                 : base(memory)
351             {
352             }
353
```

```
354                 //protected override IntPtr GetLeft(ulong node) => new
                    ↪  IntPtr(&Links[node].LeftAsTarget);
355
356                 //protected override IntPtr GetRight(ulong node) => new
                    ↪  IntPtr(&Links[node].RightAsTarget);
357
358                 //protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;
359
360                 //protected override void SetLeft(ulong node, ulong left) =>
                    ↪  Links[node].LeftAsTarget = left;
361
362                 //protected override void SetRight(ulong node, ulong right) =>
                    ↪  Links[node].RightAsTarget = right;
363
364                 //protected override void SetSize(ulong node, ulong size) =>
                    ↪  Links[node].SizeAsTarget = size;
365
366                 protected override IntPtr GetLeftPointer(ulong node) => new
                    ↪  IntPtr(&Links[node].LeftAsTarget);
367
368                 protected override IntPtr GetRightPointer(ulong node) => new
                    ↪  IntPtr(&Links[node].RightAsTarget);
369
370                 protected override ulong GetLeftValue(ulong node) => Links[node].LeftAsTarget;
371
372                 protected override ulong GetRightValue(ulong node) => Links[node].RightAsTarget;
373
374                 protected override ulong GetSize(ulong node)
375                 {
376                     var previousValue = Links[node].SizeAsTarget;
377                     //return Math.PartialRead(previousValue, 5, -5);
378                     return (previousValue & 4294967264) >> 5;
379                 }
380
381                 protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget
                    ↪  = left;
382
383                 protected override void SetRight(ulong node, ulong right) =>
                    ↪  Links[node].RightAsTarget = right;
384
385                 protected override void SetSize(ulong node, ulong size)
386                 {
387                     var previousValue = Links[node].SizeAsTarget;
388                     //var modified = Math.PartialWrite(previousValue, size, 5, -5);
389                     var modified = (previousValue & 31) | ((size & 134217727) << 5);
390                     Links[node].SizeAsTarget = modified;
391                 }
392
393                 protected override bool GetLeftIsChild(ulong node)
394                 {
395                     var previousValue = Links[node].SizeAsTarget;
396                     //return (Integer)Math.PartialRead(previousValue, 4, 1);
397                     return (previousValue & 16) >> 4 == 1UL;
398                     // TODO: Check if this is possible to use
399                     //var nodeSize = GetSize(node);
400                     //var left = GetLeftValue(node);
401                     //var leftSize = GetSizeOrZero(left);
402                     //return leftSize > 0 && nodeSize > leftSize;
403                 }
404
405                 protected override void SetLeftIsChild(ulong node, bool value)
406                 {
407                     var previousValue = Links[node].SizeAsTarget;
408                     //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 4, 1);
409                     var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
410                     Links[node].SizeAsTarget = modified;
411                 }
412
413                 protected override bool GetRightIsChild(ulong node)
414                 {
415                     var previousValue = Links[node].SizeAsTarget;
416                     //return (Integer)Math.PartialRead(previousValue, 3, 1);
417                     return (previousValue & 8) >> 3 == 1UL;
418                     // TODO: Check if this is possible to use
419                     //var nodeSize = GetSize(node);
420                     //var right = GetRightValue(node);
421                     //var rightSize = GetSizeOrZero(right);
422                     //return rightSize > 0 && nodeSize > rightSize;
```

```
            }

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

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

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

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

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

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

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

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

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

```
using System.Collections.Generic;

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;

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)
```

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

```
            => _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;

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;

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;

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;

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;

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;

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;

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;

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;
```

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

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

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

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

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

```
            _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
        }
    }
}
```

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

```
using Platform.Interfaces;

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;

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;

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;

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;

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;

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;

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

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

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

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

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

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

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

```
using System.Collections.Generic;

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

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

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

```
using System.Collections.Generic;

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

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

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

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

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

```
using System.Collections.Generic;

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

        private readonly ISynchronizedLinks<TLink> _links;

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

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

        public bool MightContain(IList<TLink> sequence)
        {
            var links = _links.Unsync;
            return _links.SyncRoot.ExecuteReadOperation(() =>
```

```
            {
                var indexed = true;
                var i = sequence.Count;
                while (--i >= 1 && (indexed =
                    !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                    sequence[i]), default))) { }
                return indexed;
            });
        }
    }
}
```

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

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

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)
    ///
    /// x*y - найти все связи между, в последовательностях любой формы, если не стоит
    ↪  ограничитель на то, что является последовательностью, а что нет,
    /// то находятся любые структуры связей, которые содержат эти элементы именно в таком
    ↪  порядке.
    ///
    /// Рост последовательности слева и справа.
    /// Поиск со звёздочкой.
    /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
    /// так же проблема может быть решена при реализации дистанционных триггеров.
    /// Нужны ли уникальные указатели вообще?
    /// Что если обращение к информации будет происходить через содержимое всегда?
    ///
    /// Писать тесты.
    ///
    ///
    /// Можно убрать зависимость от конкретной реализации Links,
    /// на зависимость от абстрактного элемента, который может быть представлен несколькими
    ↪  способами.
    ///
    /// Можно ли как-то сделать один общий интерфейс
    ///
    ///
    /// Блокчейн и/или гит для распределённой записи транзакций.
    ///
    /// </remarks>
    public partial class Sequences : ISequences<ulong> // IList<string>, IList<ulong[]> (после
    ↪  завершения реализации Sequences)
    {
        private static readonly LinksCombinedConstants<bool, ulong, long> _constants =
        ↪  Default<LinksCombinedConstants<bool, ulong, long>>.Instance;

        /// <summary>Возвращает значение ulong, обозначающее любое количество связей.</summary>
        public const ulong ZeroOrMany = ulong.MaxValue;

        public SequencesOptions<ulong> Options;
        public readonly SynchronizedLinks<ulong> Links;
        public readonly ISynchronization Sync;
```

```
61
62          public Sequences(SynchronizedLinks<ulong> links)
63              : this(links, new SequencesOptions<ulong>())
64          {
65          }
66
67          public Sequences(SynchronizedLinks<ulong> links, SequencesOptions<ulong> options)
68          {
69              Links = links;
70              Sync = links.SyncRoot;
71              Options = options;
72
73              Options.ValidateOptions();
74              Options.InitOptions(Links);
75          }
76
77          public bool IsSequence(ulong sequence)
78          {
79              return Sync.ExecuteReadOperation(() =>
80              {
81                  if (Options.UseSequenceMarker)
82                  {
83                      return Options.MarkedSequenceMatcher.IsMatched(sequence);
84                  }
85                  return !Links.Unsync.IsPartialPoint(sequence);
86              });
87          }
88
89          [MethodImpl(MethodImplOptions.AggressiveInlining)]
90          private ulong GetSequenceByElements(ulong sequence)
91          {
92              if (Options.UseSequenceMarker)
93              {
94                  return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
95              }
96              return sequence;
97          }
98
99          private ulong GetSequenceElements(ulong sequence)
100         {
101             if (Options.UseSequenceMarker)
102             {
103                 var linkContents = new UInt64Link(Links.GetLink(sequence));
104                 if (linkContents.Source == Options.SequenceMarkerLink)
105                 {
106                     return linkContents.Target;
107                 }
108                 if (linkContents.Target == Options.SequenceMarkerLink)
109                 {
110                     return linkContents.Source;
111                 }
112             }
113             return sequence;
114         }
115
116         #region Count
117
118         public ulong Count(params ulong[] sequence)
119         {
120             if (sequence.Length == 0)
121             {
122                 return Links.Count(_constants.Any, Options.SequenceMarkerLink, _constants.Any);
123             }
124             if (sequence.Length == 1) // Первая связь это адрес
125             {
126                 if (sequence[0] == _constants.Null)
127                 {
128                     return 0;
129                 }
130                 if (sequence[0] == _constants.Any)
131                 {
132                     return Count();
133                 }
134                 if (Options.UseSequenceMarker)
135                 {
136                     return Links.Count(_constants.Any, Options.SequenceMarkerLink, sequence[0]);
137                 }
138                 return Links.Exists(sequence[0]) ? 1UL : 0;
139             }
```

```
            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;
                }
                Links.EnsureEachLinkExists(sequence);
                return CreateCore(sequence);
            });
        }

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

        #endregion

```

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

```
294                 if (left != doublet)
295                 {
296                     return PartialStepRight(handler, doublet, right);
297                 }
298                 return true;
299             });
300         }
301 
302         private bool StepRight(Func<ulong, bool> handler, ulong left, ulong right) =>
    ↪  Links.Unsync.Each(left, _constants.Any, rightStep => TryStepRightUp(handler, right,
    ↪  rightStep));
303 
304         private bool TryStepRightUp(Func<ulong, bool> handler, ulong right, ulong stepFrom)
305         {
306             var upStep = stepFrom;
307             var firstSource = Links.Unsync.GetTarget(upStep);
308             while (firstSource != right && firstSource != upStep)
309             {
310                 upStep = firstSource;
311                 firstSource = Links.Unsync.GetSource(upStep);
312             }
313             if (firstSource == right)
314             {
315                 return handler(stepFrom);
316             }
317             return true;
318         }
319 
320         private bool StepLeft(Func<ulong, bool> handler, ulong left, ulong right) =>
    ↪  Links.Unsync.Each(_constants.Any, right, leftStep => TryStepLeftUp(handler, left,
    ↪  leftStep));
321 
322         private bool TryStepLeftUp(Func<ulong, bool> handler, ulong left, ulong stepFrom)
323         {
324             var upStep = stepFrom;
325             var firstTarget = Links.Unsync.GetSource(upStep);
326             while (firstTarget != left && firstTarget != upStep)
327             {
328                 upStep = firstTarget;
329                 firstTarget = Links.Unsync.GetTarget(upStep);
330             }
331             if (firstTarget == left)
332             {
333                 return handler(stepFrom);
334             }
335             return true;
336         }
337 
338         #endregion
339 
340         #region Update
341 
342         public ulong Update(ulong[] sequence, ulong[] newSequence)
343         {
344             if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
345             {
346                 return _constants.Null;
347             }
348             if (sequence.IsNullOrEmpty())
349             {
350                 return Create(newSequence);
351             }
352             if (newSequence.IsNullOrEmpty())
353             {
354                 Delete(sequence);
355                 return _constants.Null;
356             }
357             return Sync.ExecuteWriteOperation(() =>
358             {
359                 Links.EnsureEachLinkIsAnyOrExists(sequence);
360                 Links.EnsureEachLinkExists(newSequence);
361                 return UpdateCore(sequence, newSequence);
362             });
363         }
364 
365         private ulong UpdateCore(ulong[] sequence, ulong[] newSequence)
366         {
367             ulong bestVariant;
```

```
368             if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
                ↪  !sequence.EqualTo(newSequence))
369             {
370                 bestVariant = CompactCore(newSequence);
371             }
372             else
373             {
374                 bestVariant = CreateCore(newSequence);
375             }
376             // TODO: Check all options only ones before loop execution
377             // Возможно нужно две версии Each, возвращающий фактические последовательности и с
                ↪  маркером,
378             // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
                ↪  можно получить имея только фактические последовательности.
379             foreach (var variant in Each(sequence))
380             {
381                 if (variant != bestVariant)
382                 {
383                     UpdateOneCore(variant, bestVariant);
384                 }
385             }
386             return bestVariant;
387         }
388 
389         private void UpdateOneCore(ulong sequence, ulong newSequence)
390         {
391             if (Options.UseGarbageCollection)
392             {
393                 var sequenceElements = GetSequenceElements(sequence);
394                 var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
395                 var sequenceLink = GetSequenceByElements(sequenceElements);
396                 var newSequenceElements = GetSequenceElements(newSequence);
397                 var newSequenceLink = GetSequenceByElements(newSequenceElements);
398                 if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
399                 {
400                     if (sequenceLink != _constants.Null)
401                     {
402                         Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
403                     }
404                     Links.Unsync.MergeUsages(sequenceElements, newSequenceElements);
405                 }
406                 ClearGarbage(sequenceElementsContents.Source);
407                 ClearGarbage(sequenceElementsContents.Target);
408             }
409             else
410             {
411                 if (Options.UseSequenceMarker)
412                 {
413                     var sequenceElements = GetSequenceElements(sequence);
414                     var sequenceLink = GetSequenceByElements(sequenceElements);
415                     var newSequenceElements = GetSequenceElements(newSequence);
416                     var newSequenceLink = GetSequenceByElements(newSequenceElements);
417                     if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
418                     {
419                         if (sequenceLink != _constants.Null)
420                         {
421                             Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
422                         }
423                         Links.Unsync.MergeUsages(sequenceElements, newSequenceElements);
424                     }
425                 }
426                 else
427                 {
428                     if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
429                     {
430                         Links.Unsync.MergeUsages(sequence, newSequence);
431                     }
432                 }
433             }
434         }
435 
436         #endregion
437 
438         #region Delete
439 
440         public void Delete(params ulong[] sequence)
441         {
442             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);

```

```
522         #endregion
523
524         #region Garbage Collection
525
526         /// <remarks>
527         /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
            ↪  определить извне или в унаследованном классе
528         /// </remarks>
529         [MethodImpl(MethodImplOptions.AggressiveInlining)]
530         private bool IsGarbage(ulong link) => link != Options.SequenceMarkerLink &&
            ↪  !Links.Unsync.IsPartialPoint(link) && Links.Count(link) == 0;
531
532         private void ClearGarbage(ulong link)
533         {
534             if (IsGarbage(link))
535             {
536                 var contents = new UInt64Link(Links.GetLink(link));
537                 Links.Unsync.Delete(link);
538                 ClearGarbage(contents.Source);
539                 ClearGarbage(contents.Target);
540             }
541         }
542
543         #endregion
544
545         #region Walkers
546
547         public bool EachPart(Func<ulong, bool> handler, ulong sequence)
548         {
549             return Sync.ExecuteReadOperation(() =>
550             {
551                 var links = Links.Unsync;
552                 foreach (var part in Options.Walker.Walk(sequence))
553                 {
554                     if (!handler(part))
555                     {
556                         return false;
557                     }
558                 }
559                 return true;
560             });
561         }
562
563         public class Matcher : RightSequenceWalker<ulong>
564         {
565             private readonly Sequences _sequences;
566             private readonly IList<LinkIndex> _patternSequence;
567             private readonly HashSet<LinkIndex> _linksInSequence;
568             private readonly HashSet<LinkIndex> _results;
569             private readonly Func<ulong, bool> _stopableHandler;
570             private readonly HashSet<ulong> _readAsElements;
571             private int _filterPosition;
572
573             public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
                ↪  HashSet<LinkIndex> results, Func<LinkIndex, bool> stopableHandler,
                ↪  HashSet<LinkIndex> readAsElements = null)
574                 : base(sequences.Links.Unsync, new DefaultStack<ulong>())
575             {
576                 _sequences = sequences;
577                 _patternSequence = patternSequence;
578                 _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                    ↪  _constants.Any && x != ZeroOrMany));
579                 _results = results;
580                 _stopableHandler = stopableHandler;
581                 _readAsElements = readAsElements;
582             }
583
584             protected override bool IsElement(ulong link) => base.IsElement(link) ||
                ↪  (_readAsElements != null && _readAsElements.Contains(link)) ||
                ↪  _linksInSequence.Contains(link);
585
586             public bool FullMatch(LinkIndex sequenceToMatch)
587             {
588                 _filterPosition = 0;
589                 foreach (var part in Walk(sequenceToMatch))
590                 {
591                     if (!FullMatchCore(part))
592                     {
593                         break;
594                     }
```

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

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

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

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

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

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

        #endregion
    }
}
```

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

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

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

        /// <remarks>
```

```
        /// 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);
            });
        }
```

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

```
                        for (var isi = 0; isi < linkIndex; isi++)
                        {
                            innerSequence[isi] = sequence[isi];
                        }
                        for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
                        {
                            innerSequence[isi] = sequence[isi + 1];
                        }
                    }
                    innerSequence[linkIndex] = doublet;
                    Each1(handler, innerSequence);
                    return _constants.Continue;
                });
            }
        }
    }

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

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

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

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

```
            });
        }

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

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

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

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

```
                                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;
```

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

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

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

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

```
            {
                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(() =>
            {
```

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

```
771                 PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
772             }
773             return results;
774         }
775 
776         public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
777         {
778             return Sync.ExecuteReadOperation(() =>
779             {
780                 if (sequence.Length > 0)
781                 {
782                     Links.EnsureEachLinkExists(sequence);
783                     //var firstElement = sequence[0];
784                     //if (sequence.Length == 1)
785                     //{
786                     //    //results.Add(firstElement);
787                     //    return results;
788                     //}
789                     //if (sequence.Length == 2)
790                     //{
791                     //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
792                     //    //if (doublet != Doublets.Links.Null)
793                     //    //    results.Add(doublet);
794                     //    return results;
795                     //}
796                     //var lastElement = sequence[sequence.Length - 1];
797                     //Func<ulong, bool> handler = x =>
798                     //{
799                     //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
                    ↪  results.Add(x);
800                     //    return true;
801                     //};
802                     //if (sequence.Length >= 2)
803                     //    StepRight(handler, sequence[0], sequence[1]);
804                     //var last = sequence.Length - 2;
805                     //for (var i = 1; i < last; i++)
806                     //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
807                     //if (sequence.Length >= 3)
808                     //    StepLeft(handler, sequence[sequence.Length - 2],
                    ↪  sequence[sequence.Length - 1]);
809                     //////if (sequence.Length == 1)
810                     //////{
811                     //////    throw new NotImplementedException(); // all sequences, containing
                    ↪  this element?
812                     //////}
813                     //////if (sequence.Length == 2)
814                     //////{
815                     //////    var results = new List<ulong>();
816                     //////    PartialStepRight(results.Add, sequence[0], sequence[1]);
817                     //////    return results;
818                     //////}
819                     //////var matches = new List<List<ulong>>();
820                     //////var last = sequence.Length - 1;
821                     //////for (var i = 0; i < last; i++)
822                     //////{
823                     //////    var results = new List<ulong>();
824                     //////    //StepRight(results.Add, sequence[i], sequence[i + 1]);
825                     //////    PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
826                     //////    if (results.Count > 0)
827                     //////        matches.Add(results);
828                     //////    else
829                     //////        return results;
830                     //////    if (matches.Count == 2)
831                     //////    {
832                     //////        var merged = new List<ulong>();
833                     //////        for (var j = 0; j < matches[0].Count; j++)
834                     //////            for (var k = 0; k < matches[1].Count; k++)
835                     //////                CloseInnerConnections(merged.Add, matches[0][j],
                    ↪  matches[1][k]);
836                     //////        if (merged.Count > 0)
837                     //////            matches = new List<List<ulong>> { merged };
838                     //////        else
839                     //////            return new List<ulong>();
840                     //////    }
841                     //////}
842                     //////if (matches.Count > 0)
843                     //////{
```

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

```
                }
                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;
            }
        }

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

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

        private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
        ↪  HashSet<ulong> previousMatchings, long startAt)
        {
            if (startAt >= sequence.Length) // ?
            {
```

```
1226                return previousMatchings;
1227            }
1228            var secondLinkUsages = new HashSet<ulong>();
1229            AllUsagesCore(sequence[startAt], secondLinkUsages);
1230            secondLinkUsages.Add(sequence[startAt]);
1231            var matchings = new HashSet<ulong>();
1232            //for (var i = 0; i < previousMatchings.Count; i++)
1233            foreach (var secondLinkUsage in secondLinkUsages)
1234            {
1235                foreach (var previousMatching in previousMatchings)
1236                {
1237                    //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
                        ↪ secondLinkUsage);
1238                    StepRight(matchings.AddAndReturnVoid, previousMatching, secondLinkUsage);
1239                    TryStepRightUp(matchings.AddAndReturnVoid, secondLinkUsage,
                        ↪ previousMatching);
1240                    //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
                        ↪ sequence[startAt]); // почему-то эта ошибочная запись приводит к
                        ↪ желаемым результам.
1241                    PartialStepRight(matchings.AddAndReturnVoid, previousMatching,
                        ↪ secondLinkUsage);
1242                }
1243            }
1244            if (matchings.Count == 0)
1245            {
1246                return matchings;
1247            }
1248            return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
1249        }
1250
1251        private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
            ↪ links, params ulong[] sequence)
1252        {
1253            if (sequence == null)
1254            {
1255                return;
1256            }
1257            for (var i = 0; i < sequence.Length; i++)
1258            {
1259                if (sequence[i] != _constants.Any && sequence[i] != ZeroOrMany &&
                    ↪ !links.Exists(sequence[i]))
1260                {
1261                    throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                        ↪ $"patternSequence[{i}]");
1262                }
1263            }
1264        }
1265
1266        // Pattern Matching -> Key To Triggers
1267        public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1268        {
1269            return Sync.ExecuteReadOperation(() =>
1270            {
1271                patternSequence = Simplify(patternSequence);
1272                if (patternSequence.Length > 0)
1273                {
1274                    EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1275                    var uniqueSequenceElements = new HashSet<ulong>();
1276                    for (var i = 0; i < patternSequence.Length; i++)
1277                    {
1278                        if (patternSequence[i] != _constants.Any && patternSequence[i] !=
                            ↪ ZeroOrMany)
1279                        {
1280                            uniqueSequenceElements.Add(patternSequence[i]);
1281                        }
1282                    }
1283                    var results = new HashSet<ulong>();
1284                    foreach (var uniqueSequenceElement in uniqueSequenceElements)
1285                    {
1286                        AllUsagesCore(uniqueSequenceElement, results);
1287                    }
1288                    var filteredResults = new HashSet<ulong>();
1289                    var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1290                    matcher.AddAllPatternMatchedToResults(results);
1291                    return filteredResults;
1292                }
1293                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;
        }

        #endregion

        #region Walkers

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

            #region Pattern Match

            enum PatternBlockType
            {
                Undefined,
                Gap,
                Elements
            }

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

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

            #endregion
```

```
1676            public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
                ↪  HashSet<LinkIndex> results)
1677                : base(sequences.Links.Unsync, new DefaultStack<ulong>())
1678            {
1679                _sequences = sequences;
1680                _patternSequence = patternSequence;
1681                _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                    ↪  _constants.Any && x != ZeroOrMany));
1682                _results = results;
1683                _pattern = CreateDetailedPattern();
1684            }
1685
1686            protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
                ↪  base.IsElement(link);
1687
1688            public bool PatternMatch(LinkIndex sequenceToMatch)
1689            {
1690                _patternPosition = 0;
1691                _sequencePosition = 0;
1692                foreach (var part in Walk(sequenceToMatch))
1693                {
1694                    if (!PatternMatchCore(part))
1695                    {
1696                        break;
1697                    }
1698                }
1699                return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
                    ↪  - 1 && _pattern[_patternPosition].Start == 0);
1700            }
1701
1702            private List<PatternBlock> CreateDetailedPattern()
1703            {
1704                var pattern = new List<PatternBlock>();
1705                var patternBlock = new PatternBlock();
1706                for (var i = 0; i < _patternSequence.Length; i++)
1707                {
1708                    if (patternBlock.Type == PatternBlockType.Undefined)
1709                    {
1710                        if (_patternSequence[i] == _constants.Any)
1711                        {
1712                            patternBlock.Type = PatternBlockType.Gap;
1713                            patternBlock.Start = 1;
1714                            patternBlock.Stop = 1;
1715                        }
1716                        else if (_patternSequence[i] == ZeroOrMany)
1717                        {
1718                            patternBlock.Type = PatternBlockType.Gap;
1719                            patternBlock.Start = 0;
1720                            patternBlock.Stop = long.MaxValue;
1721                        }
1722                        else
1723                        {
1724                            patternBlock.Type = PatternBlockType.Elements;
1725                            patternBlock.Start = i;
1726                            patternBlock.Stop = i;
1727                        }
1728                    }
1729                    else if (patternBlock.Type == PatternBlockType.Elements)
1730                    {
1731                        if (_patternSequence[i] == _constants.Any)
1732                        {
1733                            pattern.Add(patternBlock);
1734                            patternBlock = new PatternBlock
1735                            {
1736                                Type = PatternBlockType.Gap,
1737                                Start = 1,
1738                                Stop = 1
1739                            };
1740                        }
1741                        else if (_patternSequence[i] == ZeroOrMany)
1742                        {
1743                            pattern.Add(patternBlock);
1744                            patternBlock = new PatternBlock
1745                            {
1746                                Type = PatternBlockType.Gap,
1747                                Start = 0,
1748                                Stop = long.MaxValue
1749                            };
1750                        }
1751                        else
```

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

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

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

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

        //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out
        ↪  long maximumGap)
        //{
        //    mininumGap = 0;
        //    maximumGap = 0;
        //    element = 0;
        //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
        //    {
```

```
//        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
//            mininumGap++;
//        else if (_patternSequence[_patternPosition] == ZeroOrMany)
//            maximumGap = long.MaxValue;
//        else
//            break;
//    }

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

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

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

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;

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

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

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

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

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

```
                    }
                    else
                    {
                        totalSequenceSymbolFrequencyCounter = new
                        ↪  TotalSequenceSymbolFrequencyCounter<TLink>(links);
                    }
                    var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
                    ↪  totalSequenceSymbolFrequencyCounter);
                    var compressingConverter = new CompressingConverter<TLink>(links,
                    ↪  balancedVariantConverter, doubletFrequenciesCache);
                    LinksToSequenceConverter = compressingConverter;
                }
            }
            else
            {
                if (LinksToSequenceConverter == null)
                {
                    LinksToSequenceConverter = balancedVariantConverter;
                }
            }
            if (UseIndex && Index == null)
            {
                Index = new SequenceIndex<TLink>(links);
            }
            if (Walker == null)
            {
                Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
            }
        }

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

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

```
using System.Collections.Generic;

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;

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;

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

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

        private readonly Func<TLink, bool> _isElement;

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

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

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

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

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

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

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

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

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

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;

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;
```

```

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

```

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;

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) { }
```

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

./Platform.Data.Doublets/UInt64Link.cs

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

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

```
            yield return Source;
            yield return Target;
        }

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

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

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

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

        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

```
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;

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);
```

```
16
17          public static void EnsureEachLinkExists(this ILinks<ulong> links, IList<ulong> sequence)
18          {
19              if (sequence == null)
20              {
21                  return;
22              }
23              for (var i = 0; i < sequence.Count; i++)
24              {
25                  if (!links.Exists(sequence[i]))
26                  {
27                      throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                        ↪ $"sequence[{i}]");
28                  }
29              }
30          }
31
32          public static void EnsureEachLinkIsAnyOrExists(this ILinks<ulong> links, IList<ulong>
            ↪ sequence)
33          {
34              if (sequence == null)
35              {
36                  return;
37              }
38              for (var i = 0; i < sequence.Count; i++)
39              {
40                  if (sequence[i] != Constants.Any && !links.Exists(sequence[i]))
41                  {
42                      throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                        ↪ $"sequence[{i}]");
43                  }
44              }
45          }
46
47          public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
48          {
49              if (sequence == null)
50              {
51                  return false;
52              }
53              var constants = links.Constants;
54              for (var i = 0; i < sequence.Length; i++)
55              {
56                  if (sequence[i] == constants.Any)
57                  {
58                      return true;
59                  }
60              }
61              return false;
62          }
63
64          public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
            ↪ Func<UInt64Link, bool> isElement, bool renderIndex = false, bool renderDebug = false)
65          {
66              var sb = new StringBuilder();
67              var visited = new HashSet<ulong>();
68              links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
                ↪ innerSb.Append(link.Index), renderIndex, renderDebug);
69              return sb.ToString();
70          }
71
72          public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
            ↪ Func<UInt64Link, bool> isElement, Action<StringBuilder, UInt64Link> appendElement,
            ↪ bool renderIndex = false, bool renderDebug = false)
73          {
74              var sb = new StringBuilder();
75              var visited = new HashSet<ulong>();
76              links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
                ↪ renderDebug);
77              return sb.ToString();
78          }
79
80          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)
81          {
82              if (sb == null)
```

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

./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs

```
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;

namespace Platform.Data.Doublets
{
    public class UInt64LinksTransactionsLayer : LinksDisposableDecoratorBase<ulong> //-V3073
    {
        /// <remarks>
        /// Альтернативные варианты хранения трансформации (элемента транзакции):
        ///
        /// private enum TransitionType
        /// {
        ///     Creation,
        ///     UpdateOf,
        ///     UpdateTo,
        ///     Deletion
        /// }
        ///
        /// private struct Transition
        /// {
        ///     public ulong TransactionId;
        ///     public UniqueTimestamp Timestamp;
        ///     public TransactionItemType Type;
        ///     public Link Source;
        ///     public Link Linker;
        ///     public Link Target;
        /// }
        ///
        /// Или
        ///
        /// public struct TransitionHeader
        /// {
        ///     public ulong TransactionIdCombined;
        ///     public ulong TimestampCombined;
        ///
        ///     public ulong TransactionId
        ///     {
        ///         get
        ///         {
        ///             return (ulong) mask & TransactionIdCombined;
        ///         }
        ///     }
        ///
        ///     public UniqueTimestamp Timestamp
        ///     {
        ///         get
        ///         {
        ///             return (UniqueTimestamp)mask & TransactionIdCombined;
        ///         }
        ///     }
        ///
        ///     public TransactionItemType Type
        ///     {
        ///         get
        ///         {
        ///             // Использовать по одному биту из TransactionId и Timestamp,
        ///             // для значения в 2 бита, которое представляет тип операции
        ///             throw new NotImplementedException();
        ///         }
        ///     }
        /// }
        ///
        /// private struct Transition
        /// {
        ///     public TransitionHeader Header;
        ///     public Link Source;
        ///     public Link Linker;
        ///     public Link Target;
```

```
79          /// }
80          ///
81          /// </remarks>
82          public struct Transition
83          {
84              public static readonly long Size = Structure<Transition>.Size;
85
86              public readonly ulong TransactionId;
87              public readonly UInt64Link Before;
88              public readonly UInt64Link After;
89              public readonly Timestamp Timestamp;
90
91              public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
                ↪  transactionId, UInt64Link before, UInt64Link after)
92              {
93                  TransactionId = transactionId;
94                  Before = before;
95                  After = after;
96                  Timestamp = uniqueTimestampFactory.Create();
97              }
98
99              public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
                ↪  transactionId, UInt64Link before)
100                 : this(uniqueTimestampFactory, transactionId, before, default)
101             {
102             }
103
104             public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId)
105                 : this(uniqueTimestampFactory, transactionId, default, default)
106             {
107             }
108
109             public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
                ↪  {After}";
110         }
111
112         /// <remarks>
113         /// Другие варианты реализации транзакций (атомарности):
114         ///     1. Разделение хранения значения связи ((Source Target) или (Source Linker
            ↪  Target)) и индексов.
115         ///     2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
            ↪  потребуется решить вопрос
116         ///        со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
            ↪  пересечениями идентификаторов.
117         ///
118         /// Где хранить промежуточный список транзакций?
119         ///
120         /// В оперативной памяти:
121         ///  Минусы:
122         ///     1. Может усложнить систему, если она будет функционировать самостоятельно,
123         ///     так как нужно отдельно выделять память под список трансформаций.
124         ///     2. Выделенной оперативной памяти может не хватить, в том случае,
125         ///     если транзакция использует слишком много трансформаций.
126         ///         -> Можно использовать жёсткий диск для слишком длинных транзакций.
127         ///         -> Максимальный размер списка трансформаций можно ограничить / задать
            ↪  константой.
128         ///     3. При подтверждении транзакции (Commit) все трансформации записываются разом
            ↪  создавая задержку.
129         ///
130         /// На жёстком диске:
131         ///  Минусы:
132         ///     1. Длительный отклик, на запись каждой трансформации.
133         ///     2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
134         ///         -> Это может решаться упаковкой/исключением дублирующих операций.
135         ///         -> Также это может решаться тем, что короткие транзакции вообще
136         ///            не будут записываться в случае отката.
137         ///     3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
            ↪  операции (трансформации)
138         ///        будут записаны в лог.
139         ///
140         /// </remarks>
141         public class Transaction : DisposableBase
142         {
143             private readonly Queue<Transition> _transitions;
144             private readonly UInt64LinksTransactionsLayer _layer;
145             public bool IsCommitted { get; private set; }
146             public bool IsReverted { get; private set; }
147
148             public Transaction(UInt64LinksTransactionsLayer layer)
```

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

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

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

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

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

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

```
228         private ulong _lastCommitedTransactionId;
229
230         public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
231             : base(links)
232         {
233             if (string.IsNullOrWhiteSpace(logAddress))
234             {
235                 throw new ArgumentNullException(nameof(logAddress));
236             }
237             // В первой строке файла хранится последняя закоммиченную транзакцию.
238             // При запуске это используется для проверки удачного закрытия файла лога.
239             // In the first line of the file the last committed transaction is stored.
240             // On startup, this is used to check that the log file is successfully closed.
241             var lastCommitedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
242             var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
243             if (!lastCommitedTransition.Equals(lastWrittenTransition))
244             {
245                 Dispose();
246                 throw new NotSupportedException("Database is damaged, autorecovery is not
                    ↪ supported yet.");
247             }
248             if (lastCommitedTransition.Equals(default(Transition)))
249             {
250                 FileHelpers.WriteFirst(logAddress, lastCommitedTransition);
251             }
252             _lastCommitedTransition = lastCommitedTransition;
253             // TODO: Think about a better way to calculate or store this value
254             var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
255             _lastCommitedTransactionId = allTransitions.Max(x => x.TransactionId);
256             _uniqueTimestampFactory = new UniqueTimestampFactory();
257             _logAddress = logAddress;
258             _log = FileHelpers.Append(logAddress);
259             _transitions = new Queue<Transition>();
260             _transitionsPusher = new Task(TransitionsPusher);
261             _transitionsPusher.Start();
262         }
263
264         public IList<ulong> GetLinkValue(ulong link) => Links.GetLink(link);
265
266         public override ulong Create()
267         {
268             var createdLinkIndex = Links.Create();
269             var createdLink = new UInt64Link(Links.GetLink(createdLinkIndex));
270             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                ↪ default, createdLink));
271             return createdLinkIndex;
272         }
273
274         public override ulong Update(IList<ulong> parts)
275         {
276             var linkIndex = parts[Constants.IndexPart];
277             var beforeLink = new UInt64Link(Links.GetLink(linkIndex));
278             linkIndex = Links.Update(parts);
279             var afterLink = new UInt64Link(Links.GetLink(linkIndex));
280             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                ↪ beforeLink, afterLink));
281             return linkIndex;
282         }
283
284         public override void Delete(ulong link)
285         {
286             var deletedLink = new UInt64Link(Links.GetLink(link));
287             Links.Delete(link);
288             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                ↪ deletedLink, default));
289         }
290
291         [MethodImpl(MethodImplOptions.AggressiveInlining)]
292         private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
            ↪ _transitions;
293
294         private void CommitTransition(Transition transition)
295         {
296             if (_currentTransaction != null)
297             {
298                 Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
299             }
300             var transitions = GetCurrentTransitions();
```

```
            transitions.Enqueue(transition);
        }

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

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

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

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

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

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

        private void DisposeTransitions()
        {
            try
            {
                var pusher = _transitionsPusher;
                if (pusher != null)
                {
                    _transitionsPusher = null;
                    pusher.Wait();
                }
                if (_transitions != null)
                {
                    PushTransitions();
                }
                _log.DisposeIfPossible();
                FileHelpers.WriteFirst(_logAddress, _lastCommitedTransition);
            }
            catch
            {
            }
        }

        #region DisposalBase

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

```
            if (!wasDisposed)
            {
                DisposeTransitions();
            }
            base.Dispose(manual, wasDisposed);
        }

        #endregion
    }
}
```

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

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

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

        private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;

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

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

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

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

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

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

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

```

```
24          public TLink Convert(Doublet<TLink> doublet)
25          {
26              var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
27              if (_equalityComparer.Equals(link, default))
28              {
29                  throw new ArgumentException($"Link ({doublet}) not found.", nameof(doublet));
30              }
31              var frequency = _frequencyPropertyOperator.Get(link);
32              if (_equalityComparer.Equals(frequency, default))
33              {
34                  return default;
35              }
36              var frequencyNumber = Links.GetSource(frequency);
37              return _unaryNumberToAddressConverter.Convert(frequencyNumber);
38          }
39      }
40  }
```

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

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

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

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

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

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

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

```
                _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;

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;

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;

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.");
                    }
                }
            }
        }
```

```
54
55          // 0 - null link
56          // 1 - nil character (0 character)
57          // ...
58          // 65536 (0(1) + 65535 = 65536 possible values)
59
60          [MethodImpl(MethodImplOptions.AggressiveInlining)]
61          public static ulong FromCharToLink(char character) => (ulong)character + 1;
62
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          public static char FromLinkToChar(ulong link) => (char)(link - 1);
65
66          [MethodImpl(MethodImplOptions.AggressiveInlining)]
67          public static bool IsCharLink(ulong link) => link <= MapSize;
68
69          public static string FromLinksToString(IList<ulong> linksList)
70          {
71              var sb = new StringBuilder();
72              for (int i = 0; i < linksList.Count; i++)
73              {
74                  sb.Append(FromLinkToChar(linksList[i]));
75              }
76              return sb.ToString();
77          }
78
79          public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
80          {
81              var sb = new StringBuilder();
82              if (links.Exists(link))
83              {
84                  StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
85                      x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
                        ↪  element =>
86                      {
87                          sb.Append(FromLinkToChar(element));
88                          return true;
89                      });
90              }
91              return sb.ToString();
92          }
93
94          public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
            ↪  chars.Length);
95
96          public static ulong[] FromCharsToLinkArray(char[] chars, int count)
97          {
98              // char array to ulong array
99              var linksSequence = new ulong[count];
100             for (var i = 0; i < count; i++)
101             {
102                 linksSequence[i] = FromCharToLink(chars[i]);
103             }
104             return linksSequence;
105         }
106
107         public static ulong[] FromStringToLinkArray(string sequence)
108         {
109             // char array to ulong array
110             var linksSequence = new ulong[sequence.Length];
111             for (var i = 0; i < sequence.Length; i++)
112             {
113                 linksSequence[i] = FromCharToLink(sequence[i]);
114             }
115             return linksSequence;
116         }
117
118         public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
119         {
120             var result = new List<ulong[]>();
121             var offset = 0;
122             while (offset < sequence.Length)
123             {
124                 var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
125                 var relativeLength = 1;
126                 var absoluteLength = offset + relativeLength;
127                 while (absoluteLength < sequence.Length &&
128                        currentCategory ==
                           ↪  CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
129                 {
```

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

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

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

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

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;

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;

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;

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.GetSource(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;
                }
            });

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

```
    }
}
```

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

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

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

        [Fact]
        public static void UInt32CRUDTest()
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪  ResizableDirectMemoryLinks<uint>>>())
            {
                scope.Use<ILinks<uint>>().TestCRUDOperations();
            }
        }

        [Fact]
        public static void UInt16CRUDTest()
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪  ResizableDirectMemoryLinks<ushort>>>())
            {
                scope.Use<ILinks<ushort>>().TestCRUDOperations();
            }
        }

        [Fact]
        public static void UInt8CRUDTest()
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪  ResizableDirectMemoryLinks<byte>>>())
            {
                scope.Use<ILinks<byte>>().TestCRUDOperations();
            }
        }

        private static void TestCRUDOperations<T>(this ILinks<T> links)
        {
            var constants = links.Constants;

            var equalityComparer = EqualityComparer<T>.Default;

            // Create Link
            Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));

            var setter = new Setter<T>(constants.Null);
            links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);

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

            var linkAddress = links.Create();

            var link = new Link<T>(links.GetLink(linkAddress));

            Assert.True(link.Count == 3);
            Assert.True(equalityComparer.Equals(link.Index, linkAddress));
            Assert.True(equalityComparer.Equals(link.Source, constants.Null));
            Assert.True(equalityComparer.Equals(link.Target, constants.Null));
```

```
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)
```

```
        {
            // Constants
            var constants = links.Constants;
            var equalityComparer = EqualityComparer<T>.Default;

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

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

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

            links.Update(linkAddress1, h106E, h108E);

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

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

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

            links.Update(linkAddress2, linkAddress1, h108E);

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

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

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

            links.Update(linkAddress3, linkAddress1, linkAddress2);

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

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

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

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

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

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

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

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

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

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

            // Delete link
            links.Delete(linkAddress3);

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

            var setter3 = new Setter<T>(constants.Null);
            links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);

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

        // 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();

                        var createPoint = random.NextBoolean();

```

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

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

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

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

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

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

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

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

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

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

            l2 = links.Update(l2, l1);

            links.Delete(l2);

            Global.Trash = links.Count();

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

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

    [Fact]
    public static void BasicTransactionLogTest()
    {
        using (var scope = new TempLinksTestScope(useLog: true))
        {
            var links = scope.Links;
            var l1 = links.Create();
            var l2 = links.Create();

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

            links.Delete(l1);

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

```
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
```

```
                // Увеличивает вероятность попадания информации в кэши
                for (var i = 0; i < 10; i++)
                {
                    //0 - 10 ГБ
                    //Каждые 100 МБ срез цифр

                    //links.TestGetSourceFunction();
                    //links.TestGetSourceFunctionInParallel();
                    //links.TestGetTargetFunction();
                    //links.TestGetTargetFunctionInParallel();
                    links.Create64BillionLinks();

                    links.TestRandomSearchFixed();
                    //links.Create64BillionLinksInParallel();
                    links.TestEachFunction();
                    //links.TestForeach();
                    //links.TestParallelForeach();
                }

                links.TestDeletionOfAllLinks();

            }
            catch (Exception ex)
            {
                ex.WriteToConsole();
            }
        }*/

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

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

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

            GetBaseRandomLoopOverhead(linksStep);
            GetBaseRandomLoopOverhead(linksStep);

            var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);

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

            var loops = totalLinksToCreate / linksStep;

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

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

            ConsoleHelpers.Debug();

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

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

            ConsoleHelpers.Debug();

            ConsoleHelpers.Debug("C S D");

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

```

```
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: O(0), was dependent on creation tests")]
773         public static void TestRandomSearchAll()
774         {
775             using (var scope = new TempLinksTestScope())
776             {
777                 var links = scope.Links;
778                 ulong counter = 0;
779
780                 var maxLink = links.Count();
781
782                 var iterations = links.Count();
783
784                 ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
                    ↪  links.Count());
785
786                 var sw = Stopwatch.StartNew();
787
```

```
788                 for (var i = iterations; i > 0; i--)
789                 {
790                     var linksAddressRange = new Range<ulong>(_constants.MinPossibleIndex,
                        ↪  maxLink);
791 
792                     var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
793                     var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
794 
795                     counter += links.SearchOrDefault(source, target);
796                 }
797 
798                 var elapsedTime = sw.Elapsed;
799 
800                 var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
801 
802                 ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
                    ↪  Iterations per second), c: {3}",
803                     iterations, elapsedTime, (long)iterationsPerSecond, counter);
804             }
805         }
806 
807         [Fact(Skip = "useless: 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]
```

```
        public static void TestParallelForeach()
        {
            var tempFilename = Path.GetTempFileName();

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

                long counter = 0;

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

                var sw = Stopwatch.StartNew();

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

                var elapsedTime = sw.Elapsed;

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

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

            File.Delete(tempFilename);
        }
        */

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

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

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

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

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

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

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

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

                var sw = Stopwatch.StartNew();

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

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

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

```

```
                var elapsedTime = sw.Elapsed;

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

                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);

                var unaryNumberToAddressConverter = new
                ↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
                var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
                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      }
94  }
```

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

```
1   using System;
2   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;

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

```

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

            var createResults = sequences.CreateAllVariants2(sequence);

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

            var partialSequence = new ulong[sequenceLength - 2];

            Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);

            var sw1 = Stopwatch.StartNew();
            var searchResults1 =
            ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();

            var sw2 = Stopwatch.StartNew();
            var searchResults2 =
            ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();

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

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

            //Global.Trash = searchResults3;

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

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

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

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

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

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

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

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

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

            var balancedVariant = balancedVariantConverter.Convert(sequence);

            var partialSequence = new ulong[sequenceLength - 2];

            Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);

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

```
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;
```

```
                var index = sequences.Options.Index;

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

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

                Assert.False(index.MightContain(sequence));

                index.Add(sequence);

                Assert.True(index.MightContain(sequence));
            }
        }

        /// <summary>Imported from https://raw.githubusercontent.com/wiki/Konard/LinksPlatform/%D0%9E-%D1%82%D0%BE%D0%BC%2C-%D0%BA%D0%B0%D0%BA-%D0%B2%D1%81%D1%91-%D0%BD%D0%B0%D1%87%D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>
        private static readonly string _exampleText =
            @"([english
             version](https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))

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

[![чёрное пространство, белое пространство](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png ""чёрное пространство, белое пространство"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png)

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?

[![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)

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

[![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)

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?

[![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)

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?

[![белая вертикальная линия, чёрный круг](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png ""белая вертикальная линия, чёрный круг"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

...

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


        private static readonly string _exampleLoremIpsumText =
            @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.";
```

```
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                 //    {
701                 //        // TODO: Find a solution for this case
702                 //    }
703                 //}
```

```
704
705                 for (int i = START; i < END; i++)
706                 {
707                     var first = compressor1.Create(arrays[i]);
708                     var second = compressor1.Create(arrays[i]);
709
710                     if (first == second)
711                     {
712                         compressed1[i] = first;
713                     }
714                     else
715                     {
716                         // TODO: Find a solution for this case
717                     }
718                 }
719
720                 var elapsed1 = sw1.Elapsed;
721
722                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
723
724                 var sw2 = Stopwatch.StartNew();
725
726                 for (int i = START; i < END; i++)
727                 {
728                     var first = balancedVariantConverter.Convert(arrays[i]);
729                     var second = balancedVariantConverter.Convert(arrays[i]);
730
731                     if (first == second)
732                     {
733                         compressed2[i] = first;
734                     }
735                 }
736
737                 var elapsed2 = sw2.Elapsed;
738
739                 Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
                    ↪ {elapsed2}");
740
741                 Assert.True(elapsed1 > elapsed2);
742
743                 // Checks
744                 for (int i = START; i < END; i++)
745                 {
746                     var sequence1 = compressed1[i];
747                     var sequence2 = compressed2[i];
748
749                     if (sequence1 != _constants.Null && sequence2 != _constants.Null)
750                     {
751                         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                            ↪ scope1.Links);
752
753                         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                            ↪ scope2.Links);
754
755                         //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
                            ↪ link.IsPartialPoint());
756                         //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
                            ↪ link.IsPartialPoint());
757
758                         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
                            ↪ arrays[i].Length > 3)
759                         //    Assert.False(structure1 == structure2);
760
761                         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
762                     }
763                 }
764
765                 Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
766                 Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
767
768                 Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
                    ↪ totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
                    ↪ totalCharacters}");
769
770                 Assert.True(scope1.Links.Count() <= scope2.Links.Count());
771
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
```

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

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

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

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

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

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

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

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

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

        [Fact(Skip = "Correct implementation is pending")]
        public static void CalculateAllUsagesTest()
        {
            const long sequenceLength = 3;

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

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

                var createResults = sequences.CreateAllVariants2(sequence);

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

                for (var i = 0; i < 1; i++)
                {
                    var linksTotalUsages1 = new ulong[links.Count() + 1];

                    sequences.CalculateAllUsages(linksTotalUsages1);

                    var linksTotalUsages2 = new ulong[links.Count() + 1];

                    sequences.CalculateAllUsages2(linksTotalUsages2);

                    var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
                    Assert.True(intersection1.Count == linksTotalUsages2.Length);
                }

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

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

```
using System.IO;
using Platform.Disposables;
using Platform.Data.Doublets.ResizableDirectMemory;
```

```
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Decorators;

namespace Platform.Data.Doublets.Tests
{
    public class TempLinksTestScope : DisposableBase
    {
        public readonly ILinks<ulong> MemoryAdapter;
        public readonly SynchronizedLinks<ulong> Links;
        public readonly Sequences.Sequences Sequences;
        public readonly string TempFilename;
        public readonly string TempTransactionLogFilename;
        private readonly bool _deleteFiles;

        public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool useLog = false)
            : this(new SequencesOptions<ulong>(), deleteFiles, useSequences, useLog)
        {
        }

        public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles = true, bool useSequences = false, bool useLog = false)
        {
            _deleteFiles = deleteFiles;
            TempFilename = Path.GetTempFileName();
            TempTransactionLogFilename = Path.GetTempFileName();

            var coreMemoryAdapter = new UInt64ResizableDirectMemoryLinks(TempFilename);

            MemoryAdapter = useLog ? (ILinks<ulong>)new UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) : coreMemoryAdapter;

            Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
            if (useSequences)
            {
                Sequences = new Sequences.Sequences(Links, sequencesOptions);
            }
        }

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

        public void DeleteFiles()
        {
            File.Delete(TempFilename);
            File.Delete(TempTransactionLogFilename);
        }
    }
}
```

./Platform.Data.Doublets.Tests/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);

                var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);

                Assert.Equal(originalCharacter, resultingCharacter);
            }
```

```
43          }
44
45          [Fact]
46          public static void StringAndUnicodeSequenceConvertersTest()
47          {
48              using (var scope = new TempLinksTestScope())
49              {
50                  var links = scope.Links;
51
52                  var itself = links.Constants.Itself;
53
54                  var meaningRoot = links.CreatePoint();
55                  var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
56                  var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
57                  var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
58                  var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
59                  var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);
60
61                  var powerOf2ToUnaryNumberConverter = new
                    ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
62                  var addressToUnaryNumberConverter = new
                    ↪  AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
63                  var charToUnicodeSymbolConverter = new
                    ↪  CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
                    ↪  unicodeSymbolMarker);
64
65                  var unaryNumberToAddressConverter = new
                    ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links,
                    ↪  powerOf2ToUnaryNumberConverter);
66                  var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
67                  var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                    ↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
68                  var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
                    ↪  frequencyPropertyMarker, frequencyMarker);
69                  var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
                    ↪  frequencyPropertyOperator, frequencyIncrementer);
70                  var linkToItsFrequencyNumberConverter = new
                    ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                    ↪  unaryNumberToAddressConverter);
71                  var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                    ↪  linkToItsFrequencyNumberConverter);
72                  var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                    ↪  sequenceToItsLocalElementLevelsConverter);
73
74                  var stringToUnicodeSymbolConverter = new
                    ↪  StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
                    ↪  index, optimalVariantConverter, unicodeSequenceMarker);
75
76                  var originalString = "Hello";
77
78                  var unicodeSequenceLink = stringToUnicodeSymbolConverter.Convert(originalString);
79
80                  var unicodeSymbolCriterionMatcher = new
                    ↪  UnicodeSymbolCriterionMatcher<ulong>(links, unicodeSymbolMarker);
81                  var unicodeSymbolToCharConverter = new
                    ↪  UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
                    ↪  unicodeSymbolCriterionMatcher);
82
83                  var unicodeSequenceCriterionMatcher = new
                    ↪  UnicodeSequenceCriterionMatcher<ulong>(links, unicodeSequenceMarker);
84
85                  var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
                    ↪  unicodeSymbolCriterionMatcher.IsMatched);
86
87                  var unicodeSequenceToStringConverter = new
                    ↪  UnicodeSequenceToStringConverter<ulong>(links,
                    ↪  unicodeSequenceCriterionMatcher, sequenceWalker,
                    ↪  unicodeSymbolToCharConverter);
88
89                  var resultingString =
                    ↪  unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
90
91                  Assert.Equal(originalString, resultingString);
92              }
93          }
94      }
95  }
```

Index