./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 Count(IList<TLink> restriction)
        {
            Links.EnsureInnerReferenceExists(restriction, nameof(restriction));
            return Links.Count(restriction);
        }

        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?
//                }
//            }
```

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

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

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

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

./Platform.Data.Doublets/DoubletComparer.cs

```
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(signedType);
            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))
            {
                throw new Exception("В процессе поиска по хранилищу не было найдено связей.");
            }
            return firstLink;
        }

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

        #region Paths

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

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

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

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

        #endregion

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

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

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

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

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

        /// <summary>
        /// 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))
```

```
            {
                usagesAsTarget--;
            }
            return usagesAsSource + usagesAsTarget;
        }

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

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

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

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

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

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

        /// <summary>
        /// Обновляет связь с указанными началом (Source) и концом (Target)
        /// на связь с указанными началом (NewSource) и концом (NewTarget).
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
```

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

```
618                 if (!equalityComparer.Equals(link[i], nullConstant))
619                 {
620                     return false;
621                 }
622             }
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);
```

```
                                }
                            }
                        }
                        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;
```

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

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

./Platform.Data.Doublets/LinkExtensions.cs

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

./Platform.Data.Doublets/LinksOperatorBase.cs

```
1  namespace Platform.Data.Doublets
2  {
3      public abstract class LinksOperatorBase<TLink>
4      {
5          public ILinks<TLink> Links { get; }
6          protected LinksOperatorBase(ILinks<TLink> links) => Links = links;
7      }
8  }
```

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

```
1  using System.Linq;
2  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();
```

```
44              public static readonly int SizeAsSourceOffset = Marshal.OffsetOf(typeof(Link),
                ↪  nameof(SizeAsSource)).ToInt32();
45              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                     }
```

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

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

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

```
                    }
                    var storedLinkValue = GetLinkUnsafe(index);
                    if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
                        _equalityComparer.Equals(Link.GetTarget(storedLinkValue), 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 (_equalityComparer.Equals(index, Constants.Any))
                {
                    if (_equalityComparer.Equals(source, Constants.Any) &&
                    ↪  _equalityComparer.Equals(target, Constants.Any))
                    {
                        return Each(handler, ArrayPool<TLink>.Empty);
                    }
                    else if (_equalityComparer.Equals(source, Constants.Any))
                    {
                        return _targetsTreeMethods.EachUsage(target, handler);
                    }
                    else if (_equalityComparer.Equals(target, Constants.Any))
                    {
                        return _sourcesTreeMethods.EachUsage(source, handler);
                    }
                    else //if(source != Any && target != Any)
                    {
                        var link = _sourcesTreeMethods.Search(source, target);
                        return _equalityComparer.Equals(link, Constants.Null) ?
                        ↪  Constants.Continue : handler(GetLinkStruct(link));
                    }
                }
                else
                {
                    if (!Exists(index))
                    {
                        return Constants.Continue;
                    }
                    if (_equalityComparer.Equals(source, Constants.Any) &&
                    ↪  _equalityComparer.Equals(target, Constants.Any))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    var storedLinkValue = GetLinkUnsafe(index);
                    if (!_equalityComparer.Equals(source, Constants.Any) &&
                    ↪  !_equalityComparer.Equals(target, Constants.Any))
                    {
                        if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source) &&
                            _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
                        {
                            return handler(GetLinkStruct(index));
                        }
                        return Constants.Continue;
                    }
                    var value = default(TLink);
                    if (_equalityComparer.Equals(source, Constants.Any))
                    {
                        value = target;
                    }
                    if (_equalityComparer.Equals(target, Constants.Any))
                    {
                        value = source;
                    }
                    if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
                        _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    return Constants.Continue;
                }
            }
            throw new NotSupportedException("Другие размеры и способы ограничений не
            ↪  поддерживаются.");
```

```
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         {
```

```
            if (!wasDisposed)
            {
                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
{
```

```
12      partial class ResizableDirectMemoryLinks<TLink>
13      {
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);
91                      }
92                      else
93                      {
94                          totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
95
96                          root = GetRightOrDefault(root);
97                      }
98                  }
99                  return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
100             }
101
102             public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
103             {
104                 var root = GetTreeRoot();
105                 if (EqualToZero(root))
106                 {
107                     return _constants.Continue;
108                 }
109                 TLink first = GetZero(), current = root;
110                 while (!EqualToZero(current))
111                 {
112                     var @base = GetBasePartValue(current);
113                     if (GreaterOrEqualThan(@base, link))
114                     {
115                         if (IsEquals(@base, link))
116                         {
117                             first = current;
118                         }
119                         current = GetLeftOrDefault(current);
120                     }
121                     else
122                     {
123                         current = GetRightOrDefault(current);
124                     }
125                 }
126                 if (!EqualToZero(first))
127                 {
128                     current = first;
129                     while (true)
130                     {
131                         if (IsEquals(handler(_memory.GetLinkStruct(current)), _constants.Break))
132                         {
133                             return _constants.Break;
134                         }
135                         current = GetNext(current);
136                         if (EqualToZero(current) || !IsEquals(GetBasePartValue(current), link))
137                         {
138                             break;
139                         }
140                     }
141                 }
142                 return _constants.Continue;
143             }
144
145             protected override void PrintNodeValue(TLink node, StringBuilder sb)
146             {
147                 sb.Append(' ');
148                 sb.Append((Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SourceOffset).GetValue<TLink>());
149                 sb.Append('-');
150                 sb.Append('>');
151                 sb.Append((Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.TargetOffset).GetValue<TLink>());
152             }
153         }
154
155         private class LinksSourcesTreeMethods : LinksTreeMethodsBase
156         {
157             public LinksSourcesTreeMethods(ResizableDirectMemoryLinks<TLink> memory)
158                 : base(memory)
159             {
160             }
161
162             protected override IntPtr GetLeftPointer(TLink node) =>
                ↪ Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsSourceOffset;
163
```

```
164             protected override IntPtr GetRightPointer(TLink node) =>
                ↪ Links.GetElement(LinkSizeInBytes, node) + Link.RightAsSourceOffset;
165
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     }
```

```
}
```

./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) { }
```

```
78
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                     {
```

```
                        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)
```

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

```
                _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);
                }
            }
        }

```

```
455         /// <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 
```

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

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

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

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

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

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

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

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

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

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

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

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

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

```
61
62              // TODO: Return indices range instead of references count
63              public ulong CountUsages(ulong link)
64              {
65                  var root = GetTreeRoot();
66                  var total = GetSize(root);
67                  var totalRightIgnore = 0UL;
68                  while (root != 0)
69                  {
70                      var @base = GetBasePartValue(root);
71                      if (@base <= link)
72                      {
73                          root = GetRightOrDefault(root);
74                      }
75                      else
76                      {
77                          totalRightIgnore += GetRightSize(root) + 1;
78                          root = GetLeftOrDefault(root);
79                      }
80                  }
81                  root = GetTreeRoot();
82                  var totalLeftIgnore = 0UL;
83                  while (root != 0)
84                  {
85                      var @base = GetBasePartValue(root);
86                      if (@base >= link)
87                      {
88                          root = GetLeftOrDefault(root);
89                      }
90                      else
91                      {
92                          totalLeftIgnore += GetLeftSize(root) + 1;
93                          root = GetRightOrDefault(root);
94                      }
95                  }
96                  return total - totalRightIgnore - totalLeftIgnore;
97              }
98
99              public ulong EachReference(ulong link, Func<IList<ulong>, ulong> handler)
100             {
101                 var root = GetTreeRoot();
102                 if (root == 0)
103                 {
104                     return _constants.Continue;
105                 }
106                 ulong first = 0, current = root;
107                 while (current != 0)
108                 {
109                     var @base = GetBasePartValue(current);
110                     if (@base >= link)
111                     {
112                         if (@base == link)
113                         {
114                             first = current;
115                         }
116                         current = GetLeftOrDefault(current);
117                     }
118                     else
119                     {
120                         current = GetRightOrDefault(current);
121                     }
122                 }
123                 if (first != 0)
124                 {
125                     current = first;
126                     while (true)
127                     {
128                         if (handler(_memory.GetLinkStruct(current)) == _constants.Break)
129                         {
130                             return _constants.Break;
131                         }
132                         current = GetNext(current);
133                         if (current == 0 || GetBasePartValue(current) != link)
134                         {
135                             break;
136                         }
137                     }
138                 }
139                 return _constants.Continue;
```

```
140             }
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;

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.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;

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

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

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

        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) { }

        public SynchronizedLinks(ISynchronization synchronization, ILinks<T> links)
        {
```

```
            SyncRoot = synchronization;
            Sync = this;
            Unsync = links;
            Constants = links.Constants;
        }

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

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

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

./Platform.Data.Doublets/UInt64Link.cs

```
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 UInt64Link : IEquatable<UInt64Link>, IReadOnlyList<ulong>, IList<ulong>
    {
        private static readonly LinksCombinedConstants<bool, ulong, int> _constants = Default<LinksCombinedConstants<bool, ulong, int>>.Instance;

        private const int Length = 3;

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

        public static readonly UInt64Link Null = new UInt64Link();

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

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

        public UInt64Link(ulong index, ulong source, ulong target)
```

```
        {
            Index = index;
            Source = source;
            Target = target;
        }

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

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

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

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

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

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

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

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

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

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

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

        #region IList

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

        public int Count => Length;

        public bool IsReadOnly => true;

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

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

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

./Platform.Data.Doublets/UInt64LinkExtensions.cs

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

./Platform.Data.Doublets/UInt64LinksExtensions.cs

```
1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using Platform.Singletons;
5  using Platform.Data.Constants;
6  using Platform.Data.Exceptions;
7  using Platform.Data.Doublets.Sequences;
8
9  namespace Platform.Data.Doublets
10 {
11     public static class UInt64LinksExtensions
12     {
13         public static readonly LinksCombinedConstants<bool, ulong, int> Constants =
               ↪  Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
14
15         public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
16
17         public static void EnsureEachLinkExists(this ILinks<ulong> links, IList<ulong> sequence)
18         {
```

```
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              }
```

```
            if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
            ↪  Constants.Itself)
            {
                return;
            }
            if (links.Exists(linkIndex))
            {
                if (visited.Add(linkIndex))
                {
                    sb.Append('(');
                    var link = new UInt64Link(links.GetLink(linkIndex));
                    if (renderIndex)
                    {
                        sb.Append(link.Index);
                        sb.Append(':');
                    }
                    if (link.Source == link.Index)
                    {
                        sb.Append(link.Index);
                    }
                    else
                    {
                        var source = new UInt64Link(links.GetLink(link.Source));
                        if (isElement(source))
                        {
                            appendElement(sb, source);
                        }
                        else
                        {
                            links.AppendStructure(sb, visited, source.Index, isElement,
                            ↪  appendElement, renderIndex);
                        }
                    }
                    sb.Append(' ');
                    if (link.Target == link.Index)
                    {
                        sb.Append(link.Index);
                    }
                    else
                    {
                        var target = new UInt64Link(links.GetLink(link.Target));
                        if (isElement(target))
                        {
                            appendElement(sb, target);
                        }
                        else
                        {
                            links.AppendStructure(sb, visited, target.Index, isElement,
                            ↪  appendElement, renderIndex);
                        }
                    }
                    sb.Append(')');
                }
                else
                {
                    if (renderDebug)
                    {
                        sb.Append('*');
                    }
                    sb.Append(linkIndex);
                }
            }
            else
            {
                if (renderDebug)
                {
                    sb.Append('~');
                }
                sb.Append(linkIndex);
            }
        }
    }
}
```

./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
            }
            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.Converters
{
    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.Converters
{
    public class LinkToItsFrequencyNumberConveter<TLink> : LinksOperatorBase<TLink>, IConverter<Doublet<TLink>, TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer = EqualityComparer<TLink>.Default;

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

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

        public TLink Convert(Doublet<TLink> doublet)
        {
            var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
```

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

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

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

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

        private readonly TLink[] _unaryNumberPowersOf2;

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

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

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

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

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

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

        public UnaryNumberToAddressAddOperationConverter(ILinks<TLink> links, TLink unaryOne)
            : base(links)
        {
            _unaryOne = unaryOne;
            InitUnaryToUInt64();
        }

        private void InitUnaryToUInt64()
        {
```

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

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

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

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

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

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

        private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;

        public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
        ↪  TLink> powerOf2ToUnaryNumberConverter)
            : base(links)
        {
            _unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
            for (int i = 0; i < Type<TLink>.BitsLength; i++)
            {
                _unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
            }
        }
```

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
        {
            var result = new List<ulong[]>();
            var offset = 0;
```

```
            while (offset < sequence.Length)
            {
                var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
                var relativeLength = 1;
                var absoluteLength = offset + relativeLength;
                while (absoluteLength < sequence.Length &&
                       currentCategory ==
                       ↪  CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
                {
                    relativeLength++;
                    absoluteLength++;
                }
                // char array to ulong array
                var innerSequence = new ulong[relativeLength];
                var maxLength = offset + relativeLength;
                for (var i = offset; i < maxLength; i++)
                {
                    innerSequence[i - offset] = FromCharToLink(sequence[i]);
                }
                result.Add(innerSequence);
                offset += relativeLength;
            }
            return result;
        }

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

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

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

```
19                  scope.Use<ILinks<ulong>>().TestCRUDOperations();
20              }
21          }
22
23          [Fact]
24          public static void UInt32CRUDTest()
25          {
26              using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪ ResizableDirectMemoryLinks<uint>>>())
27              {
28                  scope.Use<ILinks<uint>>().TestCRUDOperations();
29              }
30          }
31
32          [Fact]
33          public static void UInt16CRUDTest()
34          {
35              using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪ ResizableDirectMemoryLinks<ushort>>>())
36              {
37                  scope.Use<ILinks<ushort>>().TestCRUDOperations();
38              }
39          }
40
41          [Fact]
42          public static void UInt8CRUDTest()
43          {
44              using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪ ResizableDirectMemoryLinks<byte>>>())
45              {
46                  scope.Use<ILinks<byte>>().TestCRUDOperations();
47              }
48          }
49
50          private static void TestCRUDOperations<T>(this ILinks<T> links)
51          {
52              var constants = links.Constants;
53
54              var equalityComparer = EqualityComparer<T>.Default;
55
56              // Create Link
57              Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));
58
59              var setter = new Setter<T>(constants.Null);
60              links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
61
62              Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
63
64              var linkAddress = links.Create();
65
66              var link = new Link<T>(links.GetLink(linkAddress));
67
68              Assert.True(link.Count == 3);
69              Assert.True(equalityComparer.Equals(link.Index, linkAddress));
70              Assert.True(equalityComparer.Equals(link.Source, constants.Null));
71              Assert.True(equalityComparer.Equals(link.Target, constants.Null));
72
73              Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.One));
74
75              // Get first link
76              setter = new Setter<T>(constants.Null);
77              links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);
78
79              Assert.True(equalityComparer.Equals(setter.Result, linkAddress));
80
81              // Update link to reference itself
82              links.Update(linkAddress, linkAddress, linkAddress);
83
84              link = new Link<T>(links.GetLink(linkAddress));
85
86              Assert.True(equalityComparer.Equals(link.Source, linkAddress));
87              Assert.True(equalityComparer.Equals(link.Target, linkAddress));
88
89              // Update link to reference null (prepare for delete)
90              var updated = links.Update(linkAddress, constants.Null, constants.Null);
91
92              Assert.True(equalityComparer.Equals(updated, linkAddress));
93
94              link = new Link<T>(links.GetLink(linkAddress));
```

```
95
96              Assert.True(equalityComparer.Equals(link.Source, constants.Null));
97              Assert.True(equalityComparer.Equals(link.Target, constants.Null));
98
99              // Delete link
100             links.Delete(linkAddress);
101
102             Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));
103
104             setter = new Setter<T>(constants.Null);
105             links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
106
107             Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
108         }
109
110         [Fact]
111         public static void UInt64RawNumbersCRUDTest()
112         {
113             using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪  ResizableDirectMemoryLinks<ulong>>>())
114             {
115                 scope.Use<ILinks<ulong>>().TestRawNumbersCRUDOperations();
116             }
117         }
118
119         [Fact]
120         public static void UInt32RawNumbersCRUDTest()
121         {
122             using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪  ResizableDirectMemoryLinks<uint>>>())
123             {
124                 scope.Use<ILinks<uint>>().TestRawNumbersCRUDOperations();
125             }
126         }
127
128         [Fact]
129         public static void UInt16RawNumbersCRUDTest()
130         {
131             using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪  ResizableDirectMemoryLinks<ushort>>>())
132             {
133                 scope.Use<ILinks<ushort>>().TestRawNumbersCRUDOperations();
134             }
135         }
136
137         [Fact]
138         public static void UInt8RawNumbersCRUDTest()
139         {
140             using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪  ResizableDirectMemoryLinks<byte>>>())
141             {
142                 scope.Use<ILinks<byte>>().TestRawNumbersCRUDOperations();
143             }
144         }
145
146         private static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
147         {
148             // Constants
149             var constants = links.Constants;
150             var equalityComparer = EqualityComparer<T>.Default;
151
152             var h106E = new Hybrid<T>(106L, isExternal: true);
153             var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
154             var h108E = new Hybrid<T>(-108L);
155
156             Assert.Equal(106L, h106E.AbsoluteValue);
157             Assert.Equal(107L, h107E.AbsoluteValue);
158             Assert.Equal(108L, h108E.AbsoluteValue);
159
160             // Create Link (External -> External)
161             var linkAddress1 = links.Create();
162
163             links.Update(linkAddress1, h106E, h108E);
164
165             var link1 = new Link<T>(links.GetLink(linkAddress1));
166
167             Assert.True(equalityComparer.Equals(link1.Source, h106E));
168             Assert.True(equalityComparer.Equals(link1.Target, h108E));
169
```

```
            // 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]
```

```
24          public static void EqualsPerfomanceTest()
25          {
26              const int N = 1000000;
27
28              ulong x = 10;
29              ulong y = 500;
30
31              bool result = false;
32
33              var ts1 = Performance.Measure(() =>
34              {
35                  for (int i = 0; i < N; i++)
36                  {
37                      result = Equals1(x, y);
38                  }
39              });
40
41              var ts2 = Performance.Measure(() =>
42              {
43                  for (int i = 0; i < N; i++)
44                  {
45                      result = Equals2(x, y);
46                  }
47              });
48
49              var ts3 = Performance.Measure(() =>
50              {
51                  for (int i = 0; i < N; i++)
52                  {
53                      result = Equals3(x, y);
54                  }
55              });
56
57              var equalityComparer1 = EqualityComparer<ulong>.Default;
58
59              var ts4 = Performance.Measure(() =>
60              {
61                  for (int i = 0; i < N; i++)
62                  {
63                      result = equalityComparer1.Equals(x, y);
64                  }
65              });
66
67              var equalityComparer2 = new UInt64EqualityComparer();
68
69              var ts5 = Performance.Measure(() =>
70              {
71                  for (int i = 0; i < N; i++)
72                  {
73                      result = equalityComparer2.Equals(x, y);
74                  }
75              });
76
77              Func<ulong, ulong, bool> equalityComparer3 = equalityComparer2.Equals;
78
79              var ts6 = Performance.Measure(() =>
80              {
81                  for (int i = 0; i < N; i++)
82                  {
83                      result = equalityComparer3(x, y);
84                  }
85              });
86
87              var comparer = Comparer<ulong>.Default;
88
89              var ts7 = Performance.Measure(() =>
90              {
91                  for (int i = 0; i < N; i++)
92                  {
93                      result = comparer.Compare(x, y) == 0;
94                  }
95              });
96
97              Assert.True(ts2 < ts1);
98              Assert.True(ts3 < ts2);
99              Assert.True(ts5 < ts4);
100             Assert.True(ts5 < ts6);
101
102             Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {ts5} {ts6} {ts7} {result}");
```

```
        }
    }
}
```

./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

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

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

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

```

```
153                 Assert.Equal(0UL, links.Count());
154
155                 links.Unsync.DisposeIfPossible();
156
157                 var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s
                    ↪  cope.TempTransactionLogFilename);
158                 Assert.Single(transitions);
159             }
160         }
161
162         [Fact]
163         public static void TransactionUserCodeErrorNoDataSavedTest()
164         {
165             // User Code Error (Autoreverted), no data saved
166             var itself = _constants.Itself;
167
168             TempLinksTestScope lastScope = null;
169             try
170             {
171                 using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
                    ↪  useLog: true))
172                 {
173                     var links = scope.Links;
174                     var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor
                        ↪  atorBase<ulong>)links.Unsync).Links;
175                     using (var transaction = transactionsLayer.BeginTransaction())
176                     {
177                         var l1 = links.CreateAndUpdate(itself, itself);
178                         var l2 = links.CreateAndUpdate(itself, itself);
179
180                         l2 = links.Update(l2, l2, l1, l2);
181
182                         links.CreateAndUpdate(l2, itself);
183                         links.CreateAndUpdate(l2, itself);
184
185                         //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi
                            ↪  tion>(scope.TempTransactionLogFilename);
186
187                         l2 = links.Update(l2, l1);
188
189                         links.Delete(l2);
190
191                         ExceptionThrower();
192
193                         transaction.Commit();
194                     }
195
196                     Global.Trash = links.Count();
197                 }
198             }
199             catch
200             {
201                 Assert.False(lastScope == null);
202
203                 var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l
                    ↪  astScope.TempTransactionLogFilename);
204
205                 Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
                    ↪  transitions[0].After.IsNull());
206
207                 lastScope.DeleteFiles();
208             }
209         }
210
211         [Fact]
212         public static void TransactionUserCodeErrorSomeDataSavedTest()
213         {
214             // User Code Error (Autoreverted), some data saved
215             var itself = _constants.Itself;
216
217             TempLinksTestScope lastScope = null;
218             try
219             {
220                 ulong l1;
221                 ulong l2;
222
223                 using (var scope = new TempLinksTestScope(useLog: true))
224                 {
225                     var links = scope.Links;
226                     l1 = links.CreateAndUpdate(itself, itself);
```

```
227                     l2 = links.CreateAndUpdate(itself, itself);
228
229                     l2 = links.Update(l2, l2, l1, l2);
230
231                     links.CreateAndUpdate(l2, itself);
232                     links.CreateAndUpdate(l2, itself);
233
234                     links.Unsync.DisposeIfPossible();
235
236                     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
                        ↪ scope.TempTransactionLogFilename);
237                 }
238
239                 using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
                    ↪ useLog: true))
240                 {
241                     var links = scope.Links;
242                     var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
243                     using (var transaction = transactionsLayer.BeginTransaction())
244                     {
245                         l2 = links.Update(l2, l1);
246
247                         links.Delete(l2);
248
249                         ExceptionThrower();
250
251                         transaction.Commit();
252                     }
253
254                     Global.Trash = links.Count();
255                 }
256             }
257             catch
258             {
259                 Assert.False(lastScope == null);
260
261                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
                    ↪ Scope.TempTransactionLogFilename);
262
263                 lastScope.DeleteFiles();
264             }
265         }
266
267         [Fact]
268         public static void TransactionCommit()
269         {
270             var itself = _constants.Itself;
271
272             var tempDatabaseFilename = Path.GetTempFileName();
273             var tempTransactionLogFilename = Path.GetTempFileName();
274
275             // Commit
276             using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                ↪ tempTransactionLogFilename))
277             using (var links = new UInt64Links(memoryAdapter))
278             {
279                 using (var transaction = memoryAdapter.BeginTransaction())
280                 {
281                     var l1 = links.CreateAndUpdate(itself, itself);
282                     var l2 = links.CreateAndUpdate(itself, itself);
283
284                     Global.Trash = links.Update(l2, l2, l1, l2);
285
286                     links.Delete(l1);
287
288                     transaction.Commit();
289                 }
290
291                 Global.Trash = links.Count();
292             }
293
294             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                ↪ sactionLogFilename);
295         }
296
297         [Fact]
298         public static void TransactionDamage()
299         {
```

```
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);
```

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

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

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

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

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

                        links.Delete(l2);

                        ExceptionThrower();

                        transaction.Commit();
                    }

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

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

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

        [Fact]
        public static void PathsTest()
        {
            var source = _constants.SourcePart;
            var target = _constants.TargetPart;

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

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

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

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

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

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

```
444                 b = links.Update(b, a, c);
445                 c = links.Update(c, a, b);
446
447                 Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
448                 Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
449                 Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
450
451                 Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
                    ↪  "(5:(4:5 (6:5 4)) 6)");
452                 Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
                    ↪  "(6:(5:(4:5 6) 6) 4)");
453                 Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
                    ↪  "(4:(5:4 (6:5 4)) 6)");
454
455                 // TODO: Think how to build balanced syntax tree while formatting structure (eg.
                    ↪  "(4:(5:4 6) (6:5 4)") instead of "(4:(5:4 (6:5 4)) 6)"
456
457                 Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
                    ↪  "{{5}{5}{4}{6}}");
458                 Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
                    ↪  "{{5}{6}{6}{4}}");
459                 Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
                    ↪  "{{4}{5}{4}{6}}");
460             }
461         }
462
463         private static void DefaultFormatter(StringBuilder sb, ulong link)
464         {
465             sb.Append(link.ToString());
466         }
467
468         #endregion
469
470         #region Performance
471
472         /*
473         public static void RunAllPerformanceTests()
474         {
475             try
476             {
477                 links.TestLinksInSteps();
478             }
479             catch (Exception ex)
480             {
481                 ex.WriteToConsole();
482             }
483
484             return;
485
486             try
487             {
488                 //ThreadPool.SetMaxThreads(2, 2);
489
490                 // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
    ↪  результат
491                 // Также это дополнительно помогает в отладке
492                 // Увеличивает вероятность попадания информации в кэши
493                 for (var i = 0; i < 10; i++)
494                 {
495                     //0 - 10 ГБ
496                     //Каждые 100 МБ срез цифр
497
498                     //links.TestGetSourceFunction();
499                     //links.TestGetSourceFunctionInParallel();
500                     //links.TestGetTargetFunction();
501                     //links.TestGetTargetFunctionInParallel();
502                     links.Create64BillionLinks();
503
504                     links.TestRandomSearchFixed();
505                     //links.Create64BillionLinksInParallel();
506                     links.TestEachFunction();
507                     //links.TestForeach();
508                     //links.TestParallelForeach();
509                 }
510
511                 links.TestDeletionOfAllLinks();
512
513             }
514             catch (Exception ex)
```

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

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

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

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

            GetBaseRandomLoopOverhead(linksStep);
            GetBaseRandomLoopOverhead(linksStep);

            var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);

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

            var loops = totalLinksToCreate / linksStep;

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

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

            ConsoleHelpers.Debug();

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

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

            ConsoleHelpers.Debug();

            ConsoleHelpers.Debug("C S D");

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

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

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

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

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

         private static TimeSpan GetBaseRandomLoopOverhead(long loops)
         {
             return Measure(() =>
             {
```

```
            ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
            ulong result = 0;
            for (long i = 0; i < loops; i++)
            {
                var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
                var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);

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

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

            ulong counter = 0;

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

            var sw = Stopwatch.StartNew();

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

            var elapsedTime = sw.Elapsed;

            var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

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

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

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

            long counter = 0;

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

            var sw = Stopwatch.StartNew();

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

            var elapsedTime = sw.Elapsed;

            var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

            links.Delete(firstLink);

```

```
665                 ConsoleHelpers.Debug(
666                     "{0} Iterations of GetSource function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
667                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
668             }
669         }
670 
671         [Fact(Skip = "performance test")]
672         public static void TestGetTarget()
673         {
674             using (var scope = new TempLinksTestScope())
675             {
676                 var links = scope.Links;
677                 ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
                    ↪  Iterations);
678 
679                 ulong counter = 0;
680 
681                 //var firstLink = links.First();
682                 var firstLink = links.Create();
683 
684                 var sw = Stopwatch.StartNew();
685 
686                 for (ulong i = 0; i < Iterations; i++)
687                 {
688                     counter += links.GetTarget(firstLink);
689                 }
690 
691                 var elapsedTime = sw.Elapsed;
692 
693                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
694 
695                 links.Delete(firstLink);
696 
697                 ConsoleHelpers.Debug(
698                     "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
699                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
700             }
701         }
702 
703         [Fact(Skip = "performance test")]
704         public static void TestGetTargetInParallel()
705         {
706             using (var scope = new TempLinksTestScope())
707             {
708                 var links = scope.Links;
709                 ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
                    ↪  parallel.", Iterations);
710 
711                 long counter = 0;
712 
713                 //var firstLink = links.First();
714                 var firstLink = links.Create();
715 
716                 var sw = Stopwatch.StartNew();
717 
718                 Parallel.For(0, Iterations, x =>
719                 {
720                     Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
721                     //Interlocked.Increment(ref counter);
722                 });
723 
724                 var elapsedTime = sw.Elapsed;
725 
726                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
727 
728                 links.Delete(firstLink);
729 
730                 ConsoleHelpers.Debug(
731                     "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
732                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
733             }
734         }
735 
736         // TODO: Заполнить базу данных перед тестом
737         /*
738         [Fact]
```

```
739         public void TestRandomSearchFixed()
740         {
741             var tempFilename = Path.GetTempFileName();
742 
743             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪  DefaultLinksSizeStep))
744             {
745                 long iterations = 64 * 1024 * 1024 /
    ↪  Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
746 
747                 ulong counter = 0;
748                 var maxLink = links.Total;
749 
750                 ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);
751 
752                 var sw = Stopwatch.StartNew();
753 
754                 for (var i = iterations; i > 0; i--)
755                 {
756                     var source =
    ↪  RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
757                     var target =
    ↪  RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
758 
759                     counter += links.Search(source, target);
760                 }
761 
762                 var elapsedTime = sw.Elapsed;
763 
764                 var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
765 
766                 ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
    ↪  Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
    ↪  counter);
767             }
768 
769             File.Delete(tempFilename);
770         }*/
771 
772         [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
773         public static void TestRandomSearchAll()
774         {
775             using (var scope = new TempLinksTestScope())
776             {
777                 var links = scope.Links;
778                 ulong counter = 0;
779 
780                 var maxLink = links.Count();
781 
782                 var iterations = links.Count();
783 
784                 ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
                    ↪  links.Count());
785 
786                 var sw = Stopwatch.StartNew();
787 
788                 for (var i = iterations; i > 0; i--)
789                 {
790                     var linksAddressRange = new Range<ulong>(_constants.MinPossibleIndex,
                        ↪  maxLink);
791 
792                     var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
793                     var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
794 
795                     counter += links.SearchOrDefault(source, target);
796                 }
797 
798                 var elapsedTime = sw.Elapsed;
799 
800                 var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
801 
802                 ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
                    ↪  Iterations per second), c: {3}",
803                     iterations, elapsedTime, (long)iterationsPerSecond, counter);
804             }
805         }
806 
807         [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
808         public static void TestEach()
```

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

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

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

                var sw = Stopwatch.StartNew();

                links.Each(counter.IncrementAndReturnTrue);

                var elapsedTime = sw.Elapsed;

                var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;

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

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

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

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

                var sw = Stopwatch.StartNew();

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

                var elapsedTime = sw.Elapsed;

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

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

            File.Delete(tempFilename);
        }
        */

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

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

                long counter = 0;

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

                var sw = Stopwatch.StartNew();

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

                var elapsedTime = sw.Elapsed;

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

```

```
885                 ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
    ↪  {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
886             }
887
888             File.Delete(tempFilename);
889         }
890         */
891
892         [Fact(Skip = "performance test")]
893         public static void Create64BillionLinks()
894         {
895             using (var scope = new TempLinksTestScope())
896             {
897                 var links = scope.Links;
898                 var linksBeforeTest = links.Count();
899
900                 long linksToCreate = 64 * 1024 * 1024 /
                    ↪  UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
901
902                 ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
903
904                 var elapsedTime = Performance.Measure(() =>
905                 {
906                     for (long i = 0; i < linksToCreate; i++)
907                     {
908                         links.Create();
909                     }
910                 });
911
912                 var linksCreated = links.Count() - linksBeforeTest;
913                 var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
914
915                 ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
916
917                 ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
                    ↪  linksCreated, elapsedTime,
918                     (long)linksPerSecond);
919             }
920         }
921
922         [Fact(Skip = "performance test")]
923         public static void Create64BillionLinksInParallel()
924         {
925             using (var scope = new TempLinksTestScope())
926             {
927                 var links = scope.Links;
928                 var linksBeforeTest = links.Count();
929
930                 var sw = Stopwatch.StartNew();
931
932                 long linksToCreate = 64 * 1024 * 1024 /
                    ↪  UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
933
934                 ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);
935
936                 Parallel.For(0, linksToCreate, x => links.Create());
937
938                 var elapsedTime = sw.Elapsed;
939
940                 var linksCreated = links.Count() - linksBeforeTest;
941                 var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
942
943                 ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
                    ↪  linksCreated, elapsedTime,
944                     (long)linksPerSecond);
945             }
946         }
947
948         [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
949         public static void TestDeletionOfAllLinks()
950         {
951             using (var scope = new TempLinksTestScope())
952             {
953                 var links = scope.Links;
954                 var linksBeforeTest = links.Count();
955
956                 ConsoleHelpers.Debug("Deleting all links");
957
958                 var elapsedTime = Performance.Measure(links.DeleteAll);
959
```

```
960                 var linksDeleted = linksBeforeTest - links.Count();
961                 var linksPerSecond = linksDeleted / elapsedTime.TotalSeconds;
962 
963                 ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
                    ↪ linksDeleted, elapsedTime,
964                     (long)linksPerSecond);
965             }
966         }
967 
968         #endregion
969     }
970 }
```

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

```
1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4  using Xunit;
5  using Platform.Data.Doublets.Sequences;
6  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
7  using Platform.Data.Doublets.Sequences.Frequencies.Counters;
8  using Platform.Data.Doublets.Sequences.Converters;
9  using Platform.Data.Doublets.PropertyOperators;
10 using Platform.Data.Doublets.Incrementers;
11 using Platform.Data.Doublets.Converters;
12 using Platform.Data.Doublets.Sequences.Walkers;
13 using Platform.Data.Doublets.Sequences.Indexes;
14 
15 namespace Platform.Data.Doublets.Tests
16 {
17     public static class OptimalVariantSequenceTests
18     {
19         private const string SequenceExample = "зеленела зелёная зелень";
20 
21         [Fact]
22         public static void LinksBasedFrequencyStoredOptimalVariantSequenceTest()
23         {
24             using (var scope = new TempLinksTestScope(useSequences: false))
25             {
26                 var links = scope.Links;
27                 var constants = links.Constants;
28 
29                 links.UseUnicode();
30 
31                 var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);
32 
33                 var meaningRoot = links.CreatePoint();
34                 var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
35                 var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
36                 var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
                    ↪ constants.Itself);
37 
38                 var unaryNumberToAddressConveter = new
                    ↪ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
39                 var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
40                 var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                    ↪ frequencyMarker, unaryOne, unaryNumberIncrementer);
41                 var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
                    ↪ frequencyPropertyMarker, frequencyMarker);
42                 var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
                    ↪ frequencyPropertyOperator, frequencyIncrementer);
43                 var linkToItsFrequencyNumberConverter = new
                    ↪ LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                    ↪ unaryNumberToAddressConveter);
44                 var sequenceToItsLocalElementLevelsConverter = new
                    ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
                    ↪ linkToItsFrequencyNumberConverter);
45                 var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                    ↪ sequenceToItsLocalElementLevelsConverter);
46 
47                 var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
                    ↪ Walker = new LeveledSequenceWalker<ulong>(links) });
48 
49                 ExecuteTest(links, sequences, sequence,
                    ↪ sequenceToItsLocalElementLevelsConverter, index, optimalVariantConverter);
50             }
51         }
52 
53         [Fact]
```

```
54          public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
55          {
56              using (var scope = new TempLinksTestScope(useSequences: false))
57              {
58                  var links = scope.Links;
59
60                  links.UseUnicode();
61
62                  var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);
63
64                  var linksToFrequencies = new Dictionary<ulong, ulong>();
65
66                  var totalSequenceSymbolFrequencyCounter = new
                    ↪  TotalSequenceSymbolFrequencyCounter<ulong>(links);
67
68                  var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
                    ↪  totalSequenceSymbolFrequencyCounter);
69
70                  var index = new
                    ↪  CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
71                  var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
                    ↪  ncyNumberConverter<ulong>(linkFrequenciesCache);
72
73                  var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                    ↪  linkToItsFrequencyNumberConverter);
74                  var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                    ↪  sequenceToItsLocalElementLevelsConverter);
75
76                  var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
                    ↪  Walker = new LeveledSequenceWalker<ulong>(links) });
77
78                  ExecuteTest(links, sequences, sequence,
                    ↪  sequenceToItsLocalElementLevelsConverter, index, optimalVariantConverter);
79              }
80          }
81
82          private static void ExecuteTest(SynchronizedLinks<ulong> links, Sequences.Sequences
            ↪  sequences, ulong[] sequence, SequenceToItsLocalElementLevelsConverter<ulong>
            ↪  sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
            ↪  OptimalVariantConverter<ulong> optimalVariantConverter)
83          {
84              index.Add(sequence);
85
86              var optimalVariant = optimalVariantConverter.Convert(sequence);
87
88              var readSequence1 = sequences.ToList(optimalVariant);
89
90              Assert.True(sequence.SequenceEqual(readSequence1));
91          }
92      }
93  }
```

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

```
1   using System;
2   using System.Collections.Generic;
3   using System.Diagnostics;
4   using System.Linq;
5   using Xunit;
6   using Platform.Data.Sequences;
7   using Platform.Data.Doublets.Sequences.Converters;
8   using Platform.Data.Doublets.Sequences.Walkers;
9   using Platform.Data.Doublets.Sequences;
10
11  namespace Platform.Data.Doublets.Tests
12  {
13      public static class ReadSequenceTests
14      {
15          [Fact]
16          public static void ReadSequenceTest()
17          {
18              const long sequenceLength = 2000;
19
20              using (var scope = new TempLinksTestScope(useSequences: false))
21              {
22                  var links = scope.Links;
23                  var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
                    ↪  Walker = new LeveledSequenceWalker<ulong>(links) });;;
24
25                  var sequence = new ulong[sequenceLength];
```

```
                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;

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

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

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

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

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

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

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

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

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

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

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

        //    const long sequenceLength = 8;

        //    const ulong itself = LinksConstants.Itself;

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


```

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

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

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

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

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

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

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

            var balancedVariant = balancedVariantConverter.Convert(sequence);

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

            var doublet = links.GetSource(balancedVariant);

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

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

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

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

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

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

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

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

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

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

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

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

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

            index.Add(sequence);

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

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

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

Что может быть минимальным рисунком, образом, графикой? Может быть это точка? Это ли простейшая форма? Но есть ли у точки размер? Цвет? Масса? Координаты? Время существования?

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

А что если мы вообразим движение? Нужно ли время? Каким самым коротким будет путь? Что будет если этот путь зафиксировать? Запомнить след? Как две точки становятся линией? Чертой? Гранью? Разделителем? Единицей?

Можно ли замкнуть движение? Может ли это быть кругом? Можно ли замкнуть время? Или остаётся только спираль? Но что если замкнуть предел? Создать ограничение, разделение? Получится замкнутая область? Полностью отделённая от всего остального? Но что это всё остальное? Что можно делить? В каком направлении? Ничего или всё? Пустота или полнота? Начало или конец? Или может быть это единица и ноль? Дуальность? Противоположность? А что будет с кругом если у него нет размера? Будет ли круг точкой? Точка состоящая из точек?

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

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

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

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

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

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

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

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

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

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

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

...

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


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

        [Fact]
        public static void CompressionTest()
        {
            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;

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

                var sequence = new[]
                {
                    e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
                };

                var balancedVariantConverter = new BalancedVariantConverter<ulong>(links.Unsync);
                var totalSequenceSymbolFrequencyCounter = new TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
                var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync, totalSequenceSymbolFrequencyCounter);
                var compressingConverter = new CompressingConverter<ulong>(links.Unsync, balancedVariantConverter, doubletFrequenciesCache);
```

```
461
462                 var compressedVariant = compressingConverter.Convert(sequence);
463
464                 // 1: [1]       (1->1) point
465                 // 2: [2]       (2->2) point
466                 // 3: [1,2]     (1->2) doublet
467                 // 4: [1,2,1,2] (3->3) doublet
468
469                 Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
470                 Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
471                 Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
472                 Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);
473
474                 var source = _constants.SourcePart;
475                 var target = _constants.TargetPart;
476
477                 Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
478                 Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
479                 Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
480                 Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);
481
482                 // 4 - length of sequence
483                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0)
                    ↪  == sequence[0]);
484                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
                    ↪  == sequence[1]);
485                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
                    ↪  == sequence[2]);
486                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
                    ↪  == sequence[3]);
487             }
488         }
489
490         [Fact]
491         public static void CompressionEfficiencyTest()
492         {
493             var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
                ↪  StringSplitOptions.RemoveEmptyEntries);
494             var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
495             var totalCharacters = arrays.Select(x => x.Length).Sum();
496
497             using (var scope1 = new TempLinksTestScope(useSequences: true))
498             using (var scope2 = new TempLinksTestScope(useSequences: true))
499             using (var scope3 = new TempLinksTestScope(useSequences: true))
500             {
501                 scope1.Links.Unsync.UseUnicode();
502                 scope2.Links.Unsync.UseUnicode();
503                 scope3.Links.Unsync.UseUnicode();
504
505                 var balancedVariantConverter1 = new
                    ↪  BalancedVariantConverter<ulong>(scope1.Links.Unsync);
506                 var totalSequenceSymbolFrequencyCounter = new
                    ↪  TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
507                 var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
                    ↪  totalSequenceSymbolFrequencyCounter);
508                 var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
                    ↪  balancedVariantConverter1, linkFrequenciesCache1,
                    ↪  doInitialFrequenciesIncrement: false);
509
510                 var compressor2 = scope2.Sequences;
511                 var compressor3 = scope3.Sequences;
512
513                 var constants = Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
514
515                 var sequences = compressor3;
516                 //var meaningRoot = links.CreatePoint();
517                 //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
518                 //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
519                 //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
                    ↪  constants.Itself);
520
521                 //var unaryNumberToAddressConveter = new
                    ↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
522                 //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
                    ↪  unaryOne);
523                 //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                    ↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
```

```
524                 //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
                    ↪  frequencyPropertyMarker, frequencyMarker);
525                 //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
                    ↪  frequencyPropertyOperator, frequencyIncrementer);
526                 //var linkToItsFrequencyNumberConverter = new
                    ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                    ↪  unaryNumberToAddressConveter);
527
528                 var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
                    ↪  totalSequenceSymbolFrequencyCounter);
529
530                 var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
                    ↪  ncyNumberConverter<ulong>(linkFrequenciesCache3);
531
532                 var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
                    ↪  linkToItsFrequencyNumberConverter);
533                 var optimalVariantConverter = new
                    ↪  OptimalVariantConverter<ulong>(scope3.Links.Unsync,
                    ↪  sequenceToItsLocalElementLevelsConverter);
534
535                 var compressed1 = new ulong[arrays.Length];
536                 var compressed2 = new ulong[arrays.Length];
537                 var compressed3 = new ulong[arrays.Length];
538
539                 var START = 0;
540                 var END = arrays.Length;
541
542                 //for (int i = START; i < END; i++)
543                 //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
544
545                 var initialCount1 = scope2.Links.Unsync.Count();
546
547                 var sw1 = Stopwatch.StartNew();
548
549                 for (int i = START; i < END; i++)
550                 {
551                     linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
552                     compressed1[i] = compressor1.Convert(arrays[i]);
553                 }
554
555                 var elapsed1 = sw1.Elapsed;
556
557                 var balancedVariantConverter2 = new
                    ↪  BalancedVariantConverter<ulong>(scope2.Links.Unsync);
558
559                 var initialCount2 = scope2.Links.Unsync.Count();
560
561                 var sw2 = Stopwatch.StartNew();
562
563                 for (int i = START; i < END; i++)
564                 {
565                     compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
566                 }
567
568                 var elapsed2 = sw2.Elapsed;
569
570                 for (int i = START; i < END; i++)
571                 {
572                     linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
573                 }
574
575                 var initialCount3 = scope3.Links.Unsync.Count();
576
577                 var sw3 = Stopwatch.StartNew();
578
579                 for (int i = START; i < END; i++)
580                 {
581                     //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
582                     compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
583                 }
584
585                 var elapsed3 = sw3.Elapsed;
586
587                 Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
                    ↪  Optimal variant: {elapsed3}");
588
589                 // Assert.True(elapsed1 > elapsed2);
590
```

```
591                 // Checks
592                 for (int i = START; i < END; i++)
593                 {
594                     var sequence1 = compressed1[i];
595                     var sequence2 = compressed2[i];
596                     var sequence3 = compressed3[i];
597 
598                     var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                        ↪  scope1.Links.Unsync);
599 
600                     var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                        ↪  scope2.Links.Unsync);
601 
602                     var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
                        ↪  scope3.Links.Unsync);
603 
604                     var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
                        ↪  link.IsPartialPoint());
605                     var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
                        ↪  link.IsPartialPoint());
606                     var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
                        ↪  link.IsPartialPoint());
607 
608                     //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
                        ↪  arrays[i].Length > 3)
609                     //    Assert.False(structure1 == structure2);
610                     //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
                        ↪  arrays[i].Length > 3)
611                     //    Assert.False(structure3 == structure2);
612 
613                     Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
614                     Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
615                 }
616 
617                 Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
                    ↪  totalCharacters);
618                 Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
                    ↪  totalCharacters);
619                 Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
                    ↪  totalCharacters);
620 
621                 Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
                    ↪  totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) /
                    ↪  totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) /
                    ↪  totalCharacters}");
622 
623                 Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
                    ↪  scope2.Links.Unsync.Count() - initialCount2);
624                 Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
                    ↪  scope2.Links.Unsync.Count() - initialCount2);
625 
626                 var duplicateProvider1 = new
                    ↪  DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
627                 var duplicateProvider2 = new
                    ↪  DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
628                 var duplicateProvider3 = new
                    ↪  DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
629 
630                 var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
631                 var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
632                 var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
633 
634                 var duplicates1 = duplicateCounter1.Count();
635 
636                 ConsoleHelpers.Debug("------");
637 
638                 var duplicates2 = duplicateCounter2.Count();
639 
640                 ConsoleHelpers.Debug("------");
641 
642                 var duplicates3 = duplicateCounter3.Count();
643 
644                 Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
645 
646                 linkFrequenciesCache1.ValidateFrequencies();
647                 linkFrequenciesCache3.ValidateFrequencies();
648             }
649         }
```

```
650
651         [Fact]
652         public static void CompressionStabilityTest()
653         {
654             // TODO: Fix bug (do a separate test)
655             //const ulong minNumbers = 0;
656             //const ulong maxNumbers = 1000;
657
658             const ulong minNumbers = 10000;
659             const ulong maxNumbers = 12500;
660
661             var strings = new List<string>();
662
663             for (ulong i = minNumbers; i < maxNumbers; i++)
664             {
665                 strings.Add(i.ToString());
666             }
667
668             var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
669             var totalCharacters = arrays.Select(x => x.Length).Sum();
670
671             using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
                 ↪ SequencesOptions<ulong> { UseCompression = true,
                 ↪ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
672             using (var scope2 = new TempLinksTestScope(useSequences: true))
673             {
674                 scope1.Links.UseUnicode();
675                 scope2.Links.UseUnicode();
676
677                 //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
678                 var compressor1 = scope1.Sequences;
679                 var compressor2 = scope2.Sequences;
680
681                 var compressed1 = new ulong[arrays.Length];
682                 var compressed2 = new ulong[arrays.Length];
683
684                 var sw1 = Stopwatch.StartNew();
685
686                 var START = 0;
687                 var END = arrays.Length;
688
689                 // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
690                 // Stability issue starts at 10001 or 11000
691                 //for (int i = START; i < END; i++)
692                 //{
693                 //    var first = compressor1.Compress(arrays[i]);
694                 //    var second = compressor1.Compress(arrays[i]);
695
696                 //    if (first == second)
697                 //        compressed1[i] = first;
698                 //    else
699                 //    {
700                 //        // TODO: Find a solution for this case
701                 //    }
702                 //}
703
704                 for (int i = START; i < END; i++)
705                 {
706                     var first = compressor1.Create(arrays[i]);
707                     var second = compressor1.Create(arrays[i]);
708
709                     if (first == second)
710                     {
711                         compressed1[i] = first;
712                     }
713                     else
714                     {
715                         // TODO: Find a solution for this case
716                     }
717                 }
718
719                 var elapsed1 = sw1.Elapsed;
720
721                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
722
723                 var sw2 = Stopwatch.StartNew();
724
725                 for (int i = START; i < END; i++)
726                 {
727                     var first = balancedVariantConverter.Convert(arrays[i]);
```

```
728                     var second = balancedVariantConverter.Convert(arrays[i]);
729
730                     if (first == second)
731                     {
732                         compressed2[i] = first;
733                     }
734                 }
735
736                 var elapsed2 = sw2.Elapsed;
737
738                 Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
                    ↪ {elapsed2}");
739
740                 Assert.True(elapsed1 > elapsed2);
741
742                 // Checks
743                 for (int i = START; i < END; i++)
744                 {
745                     var sequence1 = compressed1[i];
746                     var sequence2 = compressed2[i];
747
748                     if (sequence1 != _constants.Null && sequence2 != _constants.Null)
749                     {
750                         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                            ↪ scope1.Links);
751
752                         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                            ↪ scope2.Links);
753
754                         //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
                            ↪ link.IsPartialPoint());
755                         //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
                            ↪ link.IsPartialPoint());
756
757                         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
                            ↪ arrays[i].Length > 3)
758                         //    Assert.False(structure1 == structure2);
759
760                         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
761                     }
762                 }
763
764                 Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
765                 Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
766
767                 Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
                    ↪ totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
                    ↪ totalCharacters}");
768
769                 Assert.True(scope1.Links.Count() <= scope2.Links.Count());
770
771                 //compressor1.ValidateFrequencies();
772             }
773         }
774
775         [Fact]
776         public static void RundomNumbersCompressionQualityTest()
777         {
778             const ulong N = 500;
779
780             //const ulong minNumbers = 10000;
781             //const ulong maxNumbers = 20000;
782
783             //var strings = new List<string>();
784
785             //for (ulong i = 0; i < N; i++)
786             //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
                    ↪ maxNumbers).ToString());
787
788             var strings = new List<string>();
789
790             for (ulong i = 0; i < N; i++)
791             {
792                 strings.Add(RandomHelpers.Default.NextUInt64().ToString());
793             }
794
795             strings = strings.Distinct().ToList();
796
797             var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
```

```
798             var totalCharacters = arrays.Select(x => x.Length).Sum();
799
800             using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
                ↪  SequencesOptions<ulong> { UseCompression = true,
                ↪  EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
801             using (var scope2 = new TempLinksTestScope(useSequences: true))
802             {
803                 scope1.Links.UseUnicode();
804                 scope2.Links.UseUnicode();
805
806                 var compressor1 = scope1.Sequences;
807                 var compressor2 = scope2.Sequences;
808
809                 var compressed1 = new ulong[arrays.Length];
810                 var compressed2 = new ulong[arrays.Length];
811
812                 var sw1 = Stopwatch.StartNew();
813
814                 var START = 0;
815                 var END = arrays.Length;
816
817                 for (int i = START; i < END; i++)
818                 {
819                     compressed1[i] = compressor1.Create(arrays[i]);
820                 }
821
822                 var elapsed1 = sw1.Elapsed;
823
824                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
825
826                 var sw2 = Stopwatch.StartNew();
827
828                 for (int i = START; i < END; i++)
829                 {
830                     compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
831                 }
832
833                 var elapsed2 = sw2.Elapsed;
834
835                 Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
                    ↪  {elapsed2}");
836
837                 Assert.True(elapsed1 > elapsed2);
838
839                 // Checks
840                 for (int i = START; i < END; i++)
841                 {
842                     var sequence1 = compressed1[i];
843                     var sequence2 = compressed2[i];
844
845                     if (sequence1 != _constants.Null && sequence2 != _constants.Null)
846                     {
847                         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                            ↪  scope1.Links);
848
849                         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                            ↪  scope2.Links);
850
851                         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
852                     }
853                 }
854
855                 Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
856                 Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
857
858                 Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
                    ↪  totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
                    ↪  totalCharacters}");
859
860                 // Can be worse than balanced variant
861                 //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
862
863                 //compressor1.ValidateFrequencies();
864             }
865         }
866
867         [Fact]
868         public static void AllTreeBreakDownAtSequencesCreationBugTest()
869         {
```

```
            // Made out of AllPossibleConnectionsTest test.

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

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

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

                var createResults = sequences.CreateAllVariants2(sequence);

                Global.Trash = createResults;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
                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.Converters;

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]));
                }
            }
```

```
            }
        }
    }
```

Index