LinksPlatform's Platform.Data.Doublets Class Library

./Converters/AddressToUnaryNumberConverter.cs

```
\texttt{
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 target = Links.Constants.Null;
            for (int i = 0; i < CachedTypeInfo<TLink>.BitsLength; i++)
            {
                if (_equalityComparer.Equals(ArithmeticHelpers.And(number, Integer<TLink>.One), Integer<TLink>.One))
                {
                    target = _equalityComparer.Equals(target, Links.Constants.Null)
                        ? _powerOf2ToUnaryNumberConverter.Convert(i)
                        : Links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
                }
                number = (Integer<TLink>)((ulong)(Integer<TLink>)number >> 1); // Should be BitwiseHelpers.ShiftRight(number, 1);
                if (_equalityComparer.Equals(number, default))
                {
                    break;
                }
            }
            return target;
        }
    }
}
```

./Converters/LinkToItsFrequencyNumberConveter.cs

```
\texttt{
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 ISpecificPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
        private readonly IConverter<TLink> _unaryNumberToAddressConverter;

        public LinkToItsFrequencyNumberConveter(
            ILinks<TLink> links,
            ISpecificPropertyOperator<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 with {doublet.Source} source and {doublet.Target} target not found.", nameof(doublet));
            }
            var frequency = _frequencyPropertyOperator.Get(link);
            if (_equalityComparer.Equals(frequency, default))
            {
                return default;
            }
            var frequencyNumber = Links.GetSource(frequency);
            var number = _unaryNumberToAddressConverter.Convert(frequencyNumber);
            return number;
        }
    }
}
}
```

./Converters/PowerOf2ToUnaryNumberConverter.cs

```
\texttt{
using System;
using System.Collections.Generic;
using Platform.Interfaces;

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)
```

```
            {
                if (power < 0 || power >= _unaryNumberPowersOf2.Length)
                {
                    throw new ArgumentOutOfRangeException(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;
            }
        }
    }
    }
```

./Converters/UnaryNumberToAddressAddOperationConverter.cs

```
    \texttt{
    using System.Collections.Generic;
    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()
            {
                _unaryToUInt64 = new Dictionary<TLink, TLink>
                {
                    { _unaryOne, Integer<TLink>.One }
                };
                var unary = _unaryOne;
                var number = Integer<TLink>.One;
                for (var i = 1; i < 64; i++)
                {
                    _unaryToUInt64.Add(unary = Links.GetOrCreate(unary, unary), number =
                     ↪   (Integer<TLink>)((Integer<TLink>)number * 2UL));
                }
            }

            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 = ArithmeticHelpers.Add(result, _unaryToUInt64[source]);
                        target = Links.GetTarget(target);
                    }
                    result = ArithmeticHelpers.Add(result, lastValue);
                    return result;
                }
            }
        }
    }
    }
```

./Converters/UnaryNumberToAddressOrOperationConverter.cs

```
    \texttt{
    using System.Collections.Generic;
    using Platform.Interfaces;
    using Platform.Reflection;
    using Platform.Numbers;

    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 < CachedTypeInfo<TLink>.BitsLength; i++)
                {
                    _unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Con ⌋
                     ↪   vert(i),
                     ↪   i);
                }
            }

            public TLink Convert(TLink sourceNumber)
            {
```

```
            var source = sourceNumber;
            var target = Links.Constants.Null;
            while (!_equalityComparer.Equals(source, Links.Constants.Null))
            {
                if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
                {
                    source = Links.Constants.Null;
                }
                else
                {
                    powerOf2Index = _unaryNumberPowerOf2Indicies[Links.GetSource(source)];
                    source = Links.GetTarget(source);
                }
                target = (Integer<TLink>)((Integer<TLink>)target | 1UL << powerOf2Index); // MathHelpers.Or(target, MathHelpers.ShiftLeft(One, powerOf2Index))
            }
            return target;
        }
    }
}
```

./Decorators/LinksCascadeDependenciesResolver.cs

```
\texttt{
using System.Collections.Generic;
using Platform.Collections.Arrays;
using Platform.Numbers;

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

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

        public override void Delete(TLink link)
        {
            EnsureNoDependenciesOnDelete(link);
            base.Delete(link);
        }

        public void EnsureNoDependenciesOnDelete(TLink link)
        {
            ulong referencesCount = (Integer<TLink>)Links.Count(Constants.Any, link);
            var references = ArrayPool.Allocate<TLink>((long)referencesCount);
            var referencesFiller = new ArrayFiller<TLink, TLink>(references, Constants.Continue);
            Links.Each(referencesFiller.AddFirstAndReturnConstant, Constants.Any, link);
            //references.Sort() // TODO: Решить необходимо ли для корректного порядка отмены операций в транзакциях
            for (var i = (long)referencesCount - 1; i >= 0; i--)
            {
                if (_equalityComparer.Equals(references[i], link))
                {
                    continue;
                }
                Links.Delete(references[i]);
            }
            ArrayPool.Free(references);
        }
    }
}
}
```

./Decorators/LinksCascadeUniquenessAndDependenciesResolver.cs

```
\texttt{
using System.Collections.Generic;
using Platform.Collections.Arrays;
using Platform.Numbers;

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

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

        protected override TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink newLinkAddress)
        {
            // TODO: Very similar to Merge (logic should be reused)
            ulong referencesAsSourceCount = (Integer<TLink>)Links.Count(Constants.Any, oldLinkAddress, Constants.Any);
            ulong referencesAsTargetCount = (Integer<TLink>)Links.Count(Constants.Any, Constants.Any, oldLinkAddress);
            var references = ArrayPool.Allocate<TLink>((long)(referencesAsSourceCount + referencesAsTargetCount));
            var referencesFiller = new ArrayFiller<TLink, TLink>(references, Constants.Continue);
            Links.Each(referencesFiller.AddFirstAndReturnConstant, Constants.Any, oldLinkAddress, Constants.Any);
            Links.Each(referencesFiller.AddFirstAndReturnConstant, Constants.Any, Constants.Any, oldLinkAddress);
            for (ulong i = 0; i < referencesAsSourceCount; i++)
            {
                var reference = references[i];
                if (!_equalityComparer.Equals(reference, oldLinkAddress))
                {
                    Links.Update(reference, newLinkAddress, Links.GetTarget(reference));
                }
            }
            for (var i = (long)referencesAsSourceCount; i < references.Length; i++)
            {
                var reference = references[i];
                if (!_equalityComparer.Equals(reference, oldLinkAddress))
                {
                    Links.Update(reference, Links.GetSource(reference), newLinkAddress);
                }
            }
            ArrayPool.Free(references);
            return base.ResolveAddressChangeConflict(oldLinkAddress, newLinkAddress);
        }
    }
}
}
```

./Decorators/LinksDecoratorBase.cs

```
\texttt{
using System;
using System.Collections.Generic;
using Platform.Data.Constants;

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

        public readonly ILinks<T> Links;

        protected LinksDecoratorBase(ILinks<T> links)
        {
            Links = links;
            Constants = links.Constants;
        }

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

        public virtual T Each(Func<IList<T>, T> handler, IList<T> restrictions) =>
        ↪   Links.Each(handler, restrictions);

        public virtual T Create() => Links.Create();

        public virtual T Update(IList<T> restrictions) => Links.Update(restrictions);

        public virtual void Delete(T link) => Links.Delete(link);
    }
}
}
```

./Decorators/LinksDependenciesValidator.cs

```
\texttt{
using System.Collections.Generic;

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

        public override T Update(IList<T> restrictions)
        {
            Links.EnsureNoDependencies(restrictions[Constants.IndexPart]);
            return base.Update(restrictions);
        }

        public override void Delete(T link)
        {
            Links.EnsureNoDependencies(link);
            base.Delete(link);
        }
    }
}
}
```

./Decorators/LinksDisposableDecoratorBase.cs

```
\texttt{
using System;
using System.Collections.Generic;
using Platform.Disposables;
using Platform.Data.Constants;

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

        public readonly ILinks<T> Links;

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

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

        public virtual T Each(Func<IList<T>, T> handler, IList<T> restrictions) =>
        ↪   Links.Each(handler, restrictions);

        public virtual T Create() => Links.Create();

        public virtual T Update(IList<T> restrictions) => Links.Update(restrictions);

        public virtual void Delete(T link) => Links.Delete(link);

        protected override bool AllowMultipleDisposeCalls => true;

        protected override void DisposeCore(bool manual, bool wasDisposed) =>
        ↪   Disposable.TryDispose(Links);
    }
}
}
```

./Decorators/LinksInnerReferenceValidator.cs

```
\texttt{
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 LinksInnerReferenceValidator<T> : LinksDecoratorBase<T>
    {
        public LinksInnerReferenceValidator(ILinks<T> links) : base(links) { }

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

        public override T Count(IList<T> restriction)
        {
            Links.EnsureInnerReferenceExists(restriction, nameof(restriction));
            return base.Count(restriction);
        }

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

        public override void Delete(T link)
        {
            // TODO: Решить считать ли такое исключением, или лишь более конкретным
            ↪   требованием?
            Links.EnsureLinkExists(link, nameof(link));
            base.Delete(link);
        }
    }
}
```

./Decorators/LinksNonExistentReferencesCreator.cs

```
\texttt{
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 LinksNonExistentReferencesCreator<T> : LinksDecoratorBase<T>
    {
        public LinksNonExistentReferencesCreator(ILinks<T> links) : base(links) { }

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

./Decorators/LinksNullToSelfReferenceResolver.cs

```
\texttt{
using System.Collections.Generic;

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

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

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

        public override TLink Update(IList<TLink> restrictions)
        {
            restrictions[Constants.SourcePart] =
            ↪   _equalityComparer.Equals(restrictions[Constants.SourcePart],
            ↪   Constants.Null) ? restrictions[Constants.IndexPart] :
            ↪   restrictions[Constants.SourcePart];
            restrictions[Constants.TargetPart] =
            ↪   _equalityComparer.Equals(restrictions[Constants.TargetPart],
            ↪   Constants.Null) ? restrictions[Constants.IndexPart] :
            ↪   restrictions[Constants.TargetPart];
            return base.Update(restrictions);
        }
    }
}
}
```

./Decorators/LinksSelfReferenceResolver.cs

```
\texttt{
using System;
using System.Collections.Generic;

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

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

        public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink>
        ↪   restrictions)
        {
            if (!_equalityComparer.Equals(Constants.Any, Constants.Itself)
             && (((restrictions.Count > Constants.IndexPart) &&
             ↪   _equalityComparer.Equals(restrictions[Constants.IndexPart],
             ↪   Constants.Itself))
             || ((restrictions.Count > Constants.SourcePart) &&
             ↪   _equalityComparer.Equals(restrictions[Constants.SourcePart],
             ↪   Constants.Itself))
             || ((restrictions.Count > Constants.TargetPart) &&
             ↪   _equalityComparer.Equals(restrictions[Constants.TargetPart],
             ↪   Constants.Itself))))
            {
                return Constants.Continue;
            }
            return base.Each(handler, restrictions);
        }

        public override TLink Update(IList<TLink> restrictions)
        {
            restrictions[Constants.SourcePart] =
            ↪   _equalityComparer.Equals(restrictions[Constants.SourcePart],
            ↪   Constants.Itself) ? restrictions[Constants.IndexPart] :
            ↪   restrictions[Constants.SourcePart];
            restrictions[Constants.TargetPart] =
            ↪   _equalityComparer.Equals(restrictions[Constants.TargetPart],
            ↪   Constants.Itself) ? restrictions[Constants.IndexPart] :
            ↪   restrictions[Constants.TargetPart];
            return base.Update(restrictions);
        }
    }
}
}
```

./Decorators/LinksUniquenessResolver.cs

```
\texttt{
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 base.Update(restrictions);
            }
            return ResolveAddressChangeConflict(restrictions[Constants.IndexPart],
            ↪   newLinkAddress);
        }

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

./Decorators/LinksUniquenessValidator.cs

```
\texttt{
using System.Collections.Generic;

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

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

./Decorators/NonNullContentsLinkDeletionResolver.cs

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

        public override void Delete(T link)
        {
            Links.Update(link, Constants.Null, Constants.Null);
            base.Delete(link);
        }
    }
}
}
```

./Decorators/UInt64Links.cs

```
\texttt{
using System;
using System.Collections.Generic;
using Platform.Collections;
using Platform.Collections.Arrays;

namespace Platform.Data.Doublets.Decorators
{
    /// <summary>
    /// Представляет объект для работы с базой данных (файлом) в формате Links
    ↪   (массива взаимосвязей).
    /// </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
    ///
    ///     AVL - высота дерева может позволить точно расчитать размер дерева, нет
    ↪   необходимости в SBT.
    ///     AVL дерево можно прошить.
    ///
    /// Текущее теоретическое ограничение на размер связей - long.MaxValue
    /// 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)
        {
            if (restrictions.IsNullOrEmpty())
            {
                return Constants.Null;
            }
           // TODO: Remove usages of these hacks (these should not be backwards compatible)
            if (restrictions.Count == 2)
            {
                return this.Merge(restrictions[0], restrictions[1]);
            }
            if (restrictions.Count == 4)
            {
                return this.UpdateOrCreateOrGet(restrictions[0], restrictions[1], restrictions[2],
                 ↪   restrictions[3]);
            }
            // TODO: Looks like this is a common type of exceptions linked with restrictions
             ↪   support
            if (restrictions.Count != 3)
            {
                throw new NotSupportedException();
            }
            var updatedLink = restrictions[Constants.IndexPart];
            this.EnsureLinkExists(updatedLink, nameof(Constants.IndexPart));
            var newSource = restrictions[Constants.SourcePart];
            this.EnsureLinkIsItselfOrExists(newSource, nameof(Constants.SourcePart));
            var newTarget = restrictions[Constants.TargetPart];
            this.EnsureLinkIsItselfOrExists(newTarget, nameof(Constants.TargetPart));
            var existedLink = Constants.Null;
            if (newSource != Constants.Itself && newTarget != Constants.Itself)
            {
                existedLink = this.SearchOrDefault(newSource, newTarget);
            }
            if (existedLink == Constants.Null)
            {
                var before = Links.GetLink(updatedLink);
               if (before[Constants.SourcePart] != newSource || before[Constants.TargetPart] !=
                ↪   newTarget)
                {
                    Links.Update(updatedLink, newSource == Constants.Itself ? updatedLink :
                     ↪   newSource,
                                              newTarget == Constants.Itself ? updatedLink : newTarget);
                }
                return updatedLink;
            }
            else
            {
                // Replace one link with another (replaced link is deleted, children are updated
                 ↪   or deleted), it is actually merge operation
                return this.Merge(updatedLink, existedLink);
            }
        }

        /// <summary>Удаляет связь с указанным индексом.</summary>
        /// <param name="link">Индекс удаляемой связи.</param>
        public override void Delete(ulong link)
        {
            this.EnsureLinkExists(link);
            Links.Update(link, Constants.Null, Constants.Null);
            var referencesCount = Links.Count(Constants.Any, link);
            if (referencesCount > 0)
            {
                var references = new ulong[referencesCount];
                var referencesFiller = new ArrayFiller<ulong, ulong>(references,
                 ↪   Constants.Continue);
                Links.Each(referencesFiller.AddFirstAndReturnConstant, Constants.Any, link);
                //references.Sort(); // TODO: Решить необходимо ли для корректного
                 ↪   порядка отмены операций в транзакциях
                for (var i = (long)referencesCount - 1; i >= 0; i--)
                {
                    if (this.Exists(references[i]))
                    {
                        Delete(references[i]);
                    }
                }
                //else
                // TODO: Определить почему здесь есть связи, которых не существует
            }
            Links.Delete(link);
        }
    }
}
```

./Decorators/UniLinks.cs

```
\texttt{
using System;
using System.Collections.Generic;
using System.Linq;
using Platform.Collections;
using Platform.Collections.Arrays;
using Platform.Collections.Lists;
using Platform.Helpers.Scopes;
using Platform.Data.Constants;
using Platform.Data.Universal;
using System.Collections.ObjectModel;

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?
            //                }
            //            }
            //        }
            //    }
            //}
            return Constants.Continue;
        }

        public TLink Trigger(IList<TLink> patternOrCondition, Func<IList<TLink>, TLink> matchHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink> substitutionHandler)
        {
            if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
            {
                return Constants.Continue;
            }
            else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO: Check if it is a correct condition
            {
                // Or it only applies to trigger without matchHandler.
                throw new NotImplementedException();
            }
            else if (!substitution.IsNullOrEmpty()) // Creation
            {
                var before = ArrayPool<TLink>.Empty;
                // Что должно означать False здесь? Остановиться (перестать идти) или пропустить (пройти мимо) или пустить (взять)?
                if (matchHandler != null && _equalityComparer.Equals(matchHandler(before), Constants.Break))
                {
                    return Constants.Break;
                }
                var after = (IList<TLink>)substitution.ToArray();
                if (_equalityComparer.Equals(after[0], default))
                {
                    var newLink = Links.Create();
                    after[0] = newLink;
                }
                if (substitution.Count == 1)
                {
                    after = Links.GetLink(substitution[0]);
                }
                else if (substitution.Count == 3)
                {
                    Links.Update(after);
                }
                else
                {
                    throw new NotSupportedException();
                }
                if (matchHandler != null)
                {
                    return substitutionHandler(before, after);
                }
                return Constants.Continue;
            }
            else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
            {
                if (patternOrCondition.Count == 1)
                {
                    var linkToDelete = patternOrCondition[0];
                    var before = Links.GetLink(linkToDelete);
                    if (matchHandler != null && _equalityComparer.Equals(matchHandler(before), Constants.Break))
                    {
                        return Constants.Break;
                    }
                    var after = ArrayPool<TLink>.Empty;
                    Links.Update(linkToDelete, Constants.Null, Constants.Null);
                    Links.Delete(linkToDelete);
                    if (matchHandler != null)
                    {
                        return substitutionHandler(before, after);
                    }
                    return Constants.Continue;
                }
                else
                {
                    throw new NotSupportedException();
                }
            }
            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;
    }
}
```

./DoubletComparer.cs

```
\texttt{
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>>
    {
        private static readonly EqualityComparer<T> _equalityComparer =
         ↪   EqualityComparer<T>.Default;

        public static readonly DoubletComparer<T> Default = new DoubletComparer<T>();

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public int GetHashCode(Doublet<T> obj) => unchecked(obj.Source.GetHashCode()
         ↪   << 15 ^ obj.Target.GetHashCode());
    }
}
```

./Doublet.cs

```
\texttt{
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);
    }
}
}
```

./Hybrid.cs

```
\texttt{
using System;
using System.Reflection;
using Platform.Reflection;
using Platform.Converters;
using Platform.Numbers;

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 => Math.Abs(Convert.ToInt64(To.Signed(Value)));

        public Hybrid(T value)
        {
            if (CachedTypeInfo<T>.IsSigned)
            {
                throw new NotSupportedException();
            }
            Value = value;
        }

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

        public Hybrid(object value, bool isExternal)
        {
            var signedType = CachedTypeInfo<T>.SignedVersion;
            var signedValue = Convert.ChangeType(value, signedType);
            var abs = typeof(MathHelpers).GetTypeInfo().GetMethod("Abs").MakeGenericM ⌋
             ↪   ethod(signedType);
            var negate = typeof(MathHelpers).GetTypeInfo().GetMethod("Negate").MakeGen ⌋
             ↪   ericMethod(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();
    }
}
```

./ILinks.cs

```
\texttt{
using Platform.Data.Constants;

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

./ILinksExtensions.cs

```
\texttt{
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.Helpers.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 =
            ↪   ArithmeticHelpers.Decrement(i))
            {
                links.Delete(i);
                if (!equalityComparer.Equals(links.Count(), ArithmeticHelpers.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 (!MathHelpers.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 = BitwiseHelpers.GetLowestBitPosition(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.
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="handler">Обработчик каждой подходящей связи.</param>
        /// <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>
        /// <returns>True, в случае если проход по связям не был прерван и False в
        ↪   обратном случае.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>,
        ↪   TLink> handler, params TLink[] restrictions)
            => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
            ↪   links.Constants.Continue);

        /// <summary>
        /// Выполняет проход по всем связям, соответствующим шаблону, вызывая
        ↪   обработчик (handler) для каждой подходящей связи.
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <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>
        /// <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>
        /// <param name="handler">Обработчик каждой подходящей связи.</param>
        /// <returns>True, в случае если проход по связям не был прерван и False в
        ↪   обратном случае.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink
        ↪   target, Func<TLink, bool> handler)
        {
            var constants = links.Constants;
            return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
            ↪   constants.Break, constants.Any, source, target);
        }

        /// <summary>
        /// Выполняет проход по всем связям, соответствующим шаблону, вызывая
        ↪   обработчик (handler) для каждой подходящей связи.
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <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>
        /// <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>
        /// <param name="handler">Обработчик каждой подходящей связи.</param>
        /// <returns>True, в случае если проход по связям не был прерван и False в
        ↪   обратном случае.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink
        ↪   target, Func<IList<TLink>, TLink> handler)
        {
            var constants = links.Constants;
            return links.Each(handler, constants.Any, source, target);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params
        ↪   TLink[] restrictions)
        {
            var constants = links.Constants;
            int listSize = (Integer<TLink>)links.Count(restrictions);
            var list = new IList<TLink>[listSize];
            if (listSize > 0)
            {
                var filler = new ArrayFiller<IList<TLink>, TLink>(list,
                ↪   links.Constants.Continue);
                links.Each(filler.AddAndReturnConstant, restrictions);
            }
            return list;
        }

        /// <summary>
        /// Возвращает значение, определяющее существует ли связь с указанными
        ↪   началом и концом в хранилище связей.
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="source">Начало связи.</param>
        /// <param name="target">Конец связи.</param>
        /// <returns>Значение, определяющее существует ли связь.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool Exists<TLink>(this ILinks<TLink> links, TLink source, TLink
        ↪   target) =>
        ↪   Comparer<TLink>.Default.Compare(links.Count(links.Constants.Any, source,
        ↪   target), default) > 0;

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
        ↪   TLink reference, string argumentName)
        {
            if (links.IsInnerReference(reference) && !links.Exists(reference))
            {
                throw new ArgumentLinkDoesNotExistsException<TLink>(reference,
                ↪   argumentName);
            }
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, IList<TLink> restrictions, string argumentName)
        {
            for (int i = 0; i < restrictions.Count; i++)
            {
                links.EnsureInnerReferenceExists(restrictions[i], argumentName);
            }
        }

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

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

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

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

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

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

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

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

        #endregion

        /// <param name="links">Хранилище связей.</param>
        public static ulong DependenciesCount<TLink>(this ILinks<TLink> links, TLink link)
        {
            var constants = links.Constants;
            var values = links.GetLink(link);
            ulong referencesAsSource = (Integer<TLink>)links.Count(constants.Any, link, constants.Any);
            var equalityComparer = EqualityComparer<TLink>.Default;
            if (equalityComparer.Equals(values[constants.SourcePart], link))
            {
                referencesAsSource--;
            }
            ulong referencesAsTarget = (Integer<TLink>)links.Count(constants.Any, constants.Any, link);
            if (equalityComparer.Equals(values[constants.TargetPart], link))
            {
                referencesAsTarget--;
        }
        return referencesAsSource + referencesAsTarget;
    }

    /// <param name="links">Хранилище связей.</param>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static bool DependenciesExist<TLink>(this ILinks<TLink> links, TLink link)
    ↪   => links.DependenciesCount(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, newSource,
    ↪   newTarget });

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

    /// <summary>
    /// Создаёт связь (если она не существовала), либо возвращает индекс
    ↪   существующей связи с указанными Source (началом) и Target (концом).
    /// </summary>
    /// <param name="links">Хранилище связей.</param>
    /// <param name="source">Индекс связи, которая является началом на
    ↪   создаваемой связи.</param>
    /// <param name="target">Индекс связи, которая является концом для
    ↪   создаваемой связи.</param>
    /// <returns>Индекс связи, с указанным Source (началом) и Target
    ↪   (концом)</returns>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source,
    ↪   TLink target)
    {
        var link = links.SearchOrDefault(source, target);
        if (EqualityComparer<TLink>.Default.Equals(link, default))
        {
            link = links.CreateAndUpdate(source, target);
        }
        return link;
    }
```

```

        /// <summary>
        /// Обновляет связь с указанными началом (Source) и концом (Target)
        /// на связь с указанными началом (NewSource) и концом (NewTarget).
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="source">Индекс связи, которая является началом
        ↪   обновляемой связи.</param>
        /// <param name="target">Индекс связи, которая является концом обновляемой
        ↪   связи.</param>
        /// <param name="newSource">Индекс связи, которая является началом связи,
        ↪   на которую выполняется обновление.</param>
        /// <param name="newTarget">Индекс связи, которая является концом связи, на
        ↪   которую выполняется обновление.</param>
        /// <returns>Индекс обновлённой связи.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink
        ↪   source, TLink target, TLink newSource, TLink newTarget)
        {
            var equalityComparer = EqualityComparer<TLink>.Default;
            var link = links.SearchOrDefault(source, target);
            if (equalityComparer.Equals(link, default))
            {
                return links.CreateAndUpdate(newSource, newTarget);
            }
            if (equalityComparer.Equals(newSource, source) &&
            ↪   equalityComparer.Equals(newTarget, target))
            {
                return link;
            }
            return links.Update(link, newSource, newTarget);
        }

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

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

        // Replace one link with another (replaced link is deleted, children are updated or
        ↪   deleted)
        public static TLink Merge<TLink>(this ILinks<TLink> links, TLink linkIndex,
        ↪   TLink newLink)
        {
            var equalityComparer = EqualityComparer<TLink>.Default;
            if (equalityComparer.Equals(linkIndex, newLink))
            {
                return newLink;
            }
            var constants = links.Constants;
            ulong referencesAsSourceCount = (Integer<TLink>)links.Count(constants.Any,
            ↪   linkIndex, constants.Any);
            ulong referencesAsTargetCount = (Integer<TLink>)links.Count(constants.Any,
            ↪   constants.Any, linkIndex);
            var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(linkIndex)) &&
            ↪   referencesAsSourceCount == 1 && referencesAsTargetCount == 1;
            if (!isStandalonePoint)
            {
                var totalReferences = referencesAsSourceCount + referencesAsTargetCount;
                if (totalReferences > 0)
                {
                    var references = ArrayPool.Allocate<TLink>((long)totalReferences);
                    var referencesFiller = new ArrayFiller<TLink, TLink>(references,
                    ↪   links.Constants.Continue);
                    links.Each(referencesFiller.AddFirstAndReturnConstant, constants.Any,
                    ↪   linkIndex, constants.Any);
                    links.Each(referencesFiller.AddFirstAndReturnConstant, constants.Any,
                    ↪   constants.Any, linkIndex);
                    for (ulong i = 0; i < referencesAsSourceCount; i++)
                    {
                        var reference = references[i];
                        if (equalityComparer.Equals(reference, linkIndex))
                        {
                            continue;
                        }

                        links.Update(reference, newLink, links.GetTarget(reference));
                    }
                    for (var i = (long)referencesAsSourceCount; i < references.Length; i++)
                    {
                        var reference = references[i];
                        if (equalityComparer.Equals(reference, linkIndex))
                        {
                            continue;
                        }

                        links.Update(reference, links.GetSource(reference), newLink);
                    }
                    ArrayPool.Free(references);
                }
            }
            links.Delete(linkIndex);
            return newLink;
        }
    }
```

```
}
}
```

./Incrementers/FrequencyIncrementer.cs

```
\texttt{
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);
        }
    }
}
}
```

./Incrementers/LinkFrequencyIncrementer.cs

```
\texttt{
using System.Collections.Generic;
using Platform.Interfaces;

namespace Platform.Data.Doublets.Incrementers
{
    public class LinkFrequencyIncrementer<TLink> : LinksOperatorBase<TLink>,
    ↪   IIncrementer<IList<TLink>>
    {
        private readonly ISpecificPropertyOperator<TLink, TLink>
        ↪   _frequencyPropertyOperator;
        private readonly IIncrementer<TLink> _frequencyIncrementer;

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

        /// <remarks>Sequence itseft is not changed, only frequency of its doublets is
        ↪   incremented.</remarks>
        public IList<TLink> Increment(IList<TLink> sequence) // TODO: May be move to
        ↪   ILinksExtensions or make SequenceDoubletsFrequencyIncrementer
        {
            for (var i = 1; i < sequence.Count; i++)
            {
                Increment(Links.GetOrCreate(sequence[i - 1], sequence[i]));
            }
            return sequence;
        }

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

./Incrementers/UnaryNumberIncrementer.cs

```
\texttt{
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));
            }
        }
    }
}
}
```

./ISynchronizedLinks.cs

```
\texttt{
using Platform.Data.Constants;

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

./Link.cs

```
\texttt{
using System;
using System.Collections;
using System.Collections.Generic;
using Platform.Exceptions;
using Platform.Ranges;
using Platform.Helpers.Singletons;
using Platform.Data.Constants;

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

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

        private const int Length = 3;

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

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

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

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

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

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

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

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

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

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

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

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

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

        public static implicit operator Link<TLink>(TLink[] linkArray) => new
        ↪   Link<TLink>(linkArray);

        public TLink[] ToArray()
        {
            var array = new TLink[Length];
            CopyTo(array, 0);
            return array;
        }

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

        #region IList

        public int Count => Length;

        public bool IsReadOnly => true;

        public TLink 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();
        }

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

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

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

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

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

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

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

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

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

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

        #endregion
    }
}
}
```

./LinkExtensions.cs

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

./LinksOperatorBase.cs

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

./obj/Debug/netstandard2.0/Platform.Data.Doublets.AssemblyInfo.cs

```
\texttt{
//------------------------------------------------------------------------------
// <auto-generated>
//     Generated by the MSBuild WriteCodeFragment class.
// </auto-generated>
//------------------------------------------------------------------------------

using System;
using System.Reflection;

[assembly: System.Reflection.AssemblyConfigurationAttribute("Debug")]
[assembly: System.Reflection.AssemblyCopyrightAttribute("Konstantin Diachenko")]
[assembly: System.Reflection.AssemblyDescriptionAttribute("LinksPlatform\'s
 ↪   Platform.Data.Doublets Class Library")]
[assembly: System.Reflection.AssemblyFileVersionAttribute("0.0.1.0")]
[assembly: System.Reflection.AssemblyInformationalVersionAttribute("0.0.1")]
[assembly: System.Reflection.AssemblyTitleAttribute("Platform.Data.Doublets")]
[assembly: System.Reflection.AssemblyVersionAttribute("0.0.1.0")]
}
```

./PropertyOperators/DefaultLinkPropertyOperator.cs

```
\texttt{
using System.Linq;
using System.Collections.Generic;
using Platform.Interfaces;

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

        public DefaultLinkPropertyOperator(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;
            }
            var value = Links.GetTarget(valueLink[Links.Constants.IndexPart]);
            return value;
        }

        public void SetValue(TLink @object, TLink property, TLink value)
        {
            var objectProperty = Links.GetOrCreate(@object, property);
            Links.DeleteMany(Links.All(Links.Constants.Any, objectProperty).Select(link =>
            ↪   link[Links.Constants.IndexPart]).ToList());
            Links.GetOrCreate(objectProperty, value);
        }
    }
}
}
```

./PropertyOperators/FrequencyPropertyOperator.cs

```
\texttt{
using System.Collections.Generic;
using Platform.Interfaces;

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

        private readonly TLink _frequencyPropertyMarker;
        private readonly TLink _frequencyMarker;

        public FrequencyPropertyOperator(ILinks<TLink> links, TLink
        ↪   frequencyPropertyMarker, TLink frequencyMarker) : base(links)
        {
            _frequencyPropertyMarker = frequencyPropertyMarker;
            _frequencyMarker = frequencyMarker;
        }

        public TLink Get(TLink link)
        {
            var property = Links.SearchOrDefault(link, _frequencyPropertyMarker);
            var container = GetContainer(property);
            var frequency = GetFrequency(container);
            return frequency;
        }

        private TLink GetContainer(TLink property)
        {
            var frequencyContainer = default(TLink);
            if (_equalityComparer.Equals(property, default))
            {
                return frequencyContainer;
            }
            Links.Each(candidate =>
            {
                var candidateTarget = Links.GetTarget(candidate);
                var frequencyTarget = Links.GetTarget(candidateTarget);
                if (_equalityComparer.Equals(frequencyTarget, _frequencyMarker))
                {
                    frequencyContainer = Links.GetIndex(candidate);
                    return Links.Constants.Break;
                }
                return Links.Constants.Continue;
            }, Links.Constants.Any, property, Links.Constants.Any);
            return frequencyContainer;
        }

        private TLink GetFrequency(TLink container) =>
        ↪   _equalityComparer.Equals(container, default) ? default :
        ↪   Links.GetTarget(container);

        public void Set(TLink link, TLink frequency)
        {
            var property = Links.GetOrCreate(link, _frequencyPropertyMarker);
            var container = GetContainer(property);
            if (_equalityComparer.Equals(container, default))
            {
                Links.GetOrCreate(property, frequency);
            }
            else
            {
                Links.Update(container, property, frequency);
            }
        }
    }
}
}
```

./ResizableDirectMemory/ResizableDirectMemoryLinks.cs

```
\texttt{
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using Platform.Disposables;
using Platform.Helpers.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.ArithmeticHelpers;

#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 = StructureHelpers.SizeOf<Link>();

        public static readonly int LinkHeaderSizeInBytes = StructureHelpers.SizeOf<LinksHeader>();

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

        private struct Link
        {
            public static readonly int SourceOffset = Marshal.OffsetOf(typeof(Link), nameof(Source)).ToInt32();
            public static readonly int TargetOffset = Marshal.OffsetOf(typeof(Link), nameof(Target)).ToInt32();
            public static readonly int LeftAsSourceOffset = Marshal.OffsetOf(typeof(Link), nameof(LeftAsSource)).ToInt32();
            public static readonly int RightAsSourceOffset = Marshal.OffsetOf(typeof(Link), nameof(RightAsSource)).ToInt32();
            public static readonly int SizeAsSourceOffset = Marshal.OffsetOf(typeof(Link), nameof(SizeAsSource)).ToInt32();
            public static readonly int LeftAsTargetOffset = Marshal.OffsetOf(typeof(Link), nameof(LeftAsTarget)).ToInt32();
            public static readonly int RightAsTargetOffset = Marshal.OffsetOf(typeof(Link), nameof(RightAsTarget)).ToInt32();
            public static readonly int SizeAsTargetOffset = Marshal.OffsetOf(typeof(Link), nameof(SizeAsTarget)).ToInt32();

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

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

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

        private struct LinksHeader
        {
            public static readonly int AllocatedLinksOffset = Marshal.OffsetOf(typeof(LinksHeader), nameof(AllocatedLinks)).ToInt32();
            public static readonly int ReservedLinksOffset =
             ↪  Marshal.OffsetOf(typeof(LinksHeader), nameof(ReservedLinks)).ToInt32();
            public static readonly int FreeLinksOffset = Marshal.OffsetOf(typeof(LinksHeader),
             ↪  nameof(FreeLinks)).ToInt32();
            public static readonly int FirstFreeLinkOffset =
             ↪  Marshal.OffsetOf(typeof(LinksHeader), nameof(FirstFreeLink)).ToInt32();
            public static readonly int FirstAsSourceOffset =
             ↪  Marshal.OffsetOf(typeof(LinksHeader), nameof(FirstAsSource)).ToInt32();
            public static readonly int FirstAsTargetOffset =
             ↪  Marshal.OffsetOf(typeof(LinksHeader), nameof(FirstAsTarget)).ToInt32();
            public static readonly int LastFreeLinkOffset =
             ↪  Marshal.OffsetOf(typeof(LinksHeader), nameof(LastFreeLink)).ToInt32();

            public TLink AllocatedLinks;
            public TLink ReservedLinks;
            public TLink FreeLinks;
            public TLink FirstFreeLink;
            public TLink FirstAsSource;
            public TLink FirstAsTarget;
            public TLink LastFreeLink;
            public TLink Reserved8;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetAllocatedLinks(IntPtr pointer) => (pointer +
             ↪  AllocatedLinksOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetReservedLinks(IntPtr pointer) => (pointer +
             ↪  ReservedLinksOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetFreeLinks(IntPtr pointer) => (pointer +
             ↪  FreeLinksOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetFirstFreeLink(IntPtr pointer) => (pointer +
             ↪  FirstFreeLinkOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetFirstAsSource(IntPtr pointer) => (pointer +
             ↪  FirstAsSourceOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetFirstAsTarget(IntPtr pointer) => (pointer +
             ↪  FirstAsTargetOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetLastFreeLink(IntPtr pointer) => (pointer +
             ↪  LastFreeLinkOffset).GetValue<TLink>();

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static IntPtr GetFirstAsSourcePointer(IntPtr pointer) => pointer +
             ↪  FirstAsSourceOffset;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static IntPtr GetFirstAsTargetPointer(IntPtr pointer) => pointer +
             ↪  FirstAsTargetOffset;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetAllocatedLinks(IntPtr pointer, TLink value) => (pointer +
             ↪  AllocatedLinksOffset).SetValue(value);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetReservedLinks(IntPtr pointer, TLink value) => (pointer +
             ↪  ReservedLinksOffset).SetValue(value);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetFreeLinks(IntPtr pointer, TLink value) => (pointer +
             ↪  FreeLinksOffset).SetValue(value);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetFirstFreeLink(IntPtr pointer, TLink value) => (pointer +
             ↪  FirstFreeLinkOffset).SetValue(value);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetFirstAsSource(IntPtr pointer, TLink value) => (pointer +
             ↪  FirstAsSourceOffset).SetValue(value);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetFirstAsTarget(IntPtr pointer, TLink value) => (pointer +
             ↪  FirstAsTargetOffset).SetValue(value);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetLastFreeLink(IntPtr pointer, TLink value) => (pointer +
             ↪  LastFreeLinkOffset).SetValue(value);
        }

        private readonly long _memoryReservationStep;

        private readonly IResizableDirectMemory _memory;
        private IntPtr _header;
        private IntPtr _links;

        private LinksTargetsTreeMethods _targetsTreeMethods;
        private LinksSourcesTreeMethods _sourcesTreeMethods;

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

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

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

        public ResizableDirectMemoryLinks(string address)
            : this(address, DefaultLinksSizeStep)
        {
        }

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Count(IList<TLink> restrictions)
        {
            // Если нет ограничений, тогда возвращаем общее число связей находящихся в
            ↪   хранилище.
            if (restrictions.Count == 0)
            {
                return Total;
            }
            if (restrictions.Count == 1)
            {
                var index = restrictions[Constants.IndexPart];
                if (_equalityComparer.Equals(index, Constants.Any))
                {
                    return Total;
                }
                return Exists(index) ? Integer<TLink>.One : Integer<TLink>.Zero;
            }
            if (restrictions.Count == 2)
            {
                var index = restrictions[Constants.IndexPart];
                var value = restrictions[1];
                if (_equalityComparer.Equals(index, Constants.Any))
                {
                    if (_equalityComparer.Equals(value, Constants.Any))
                    {
                        return Total; // Any - как отсутствие ограничения
                    }
                    return Add(_sourcesTreeMethods.CalculateReferences(value),
                    ↪   _targetsTreeMethods.CalculateReferences(value));
                }
                else
                {
                    if (!Exists(index))
                    {
                        return Integer<TLink>.Zero;
                    }
                    if (_equalityComparer.Equals(value, Constants.Any))
                    {
                        return Integer<TLink>.One;
                    }
                    var storedLinkValue = GetLinkUnsafe(index);
                    if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
                        _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
                    {
                        return Integer<TLink>.One;
                    }
                    return Integer<TLink>.Zero;
                }
            }
            if (restrictions.Count == 3)
            {
                var index = restrictions[Constants.IndexPart];
                var source = restrictions[Constants.SourcePart];
                var target = restrictions[Constants.TargetPart];

                if (_equalityComparer.Equals(index, Constants.Any))
                {
                    if (_equalityComparer.Equals(source, Constants.Any) &&
                    ↪   _equalityComparer.Equals(target, Constants.Any))
                    {
                        return Total;
                    }
                    else if (_equalityComparer.Equals(source, Constants.Any))
                    {
                        return _targetsTreeMethods.CalculateReferences(target);
                    }
                    else if (_equalityComparer.Equals(target, Constants.Any))
                    {
                        return _sourcesTreeMethods.CalculateReferences(source);
                    }
                    else //if(source != Any && target != Any)
                    {
                        // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
                        var link = _sourcesTreeMethods.Search(source, target);
                        return _equalityComparer.Equals(link, Constants.Null) ?
                        ↪   Integer<TLink>.Zero : Integer<TLink>.One;
                    }
                }
                else
                {
                    if (!Exists(index))
                    {
                        return Integer<TLink>.Zero;
                    }
                    if (_equalityComparer.Equals(source, Constants.Any) &&
                    ↪   _equalityComparer.Equals(target, Constants.Any))
                    {
                        return Integer<TLink>.One;
                    }
                    var storedLinkValue = GetLinkUnsafe(index);
                    if (!_equalityComparer.Equals(source, Constants.Any) &&
                    ↪   !_equalityComparer.Equals(target, Constants.Any))
                    {
                        if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source)
                        ↪   &&
                            _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
                        {
                            return Integer<TLink>.One;
                        }
                        return Integer<TLink>.Zero;
                    }
                    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.EachReference(target, handler);
                    }
                    else if (_equalityComparer.Equals(target, Constants.Any))
                    {
                        return _sourcesTreeMethods.EachReference(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.");
        }

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

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

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

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

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

        /// <remarks>
        /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том
        ↪   случае, если адрес реально поменялся
        ///
        /// Указатель this.links может быть в том же месте,
        /// так как 0-я связь не используется и имеет такой же размер как Header,
        /// поэтому header размещается в том же месте, что и 0-я связь
        /// </remarks>
        private void SetPointers(IDirectMemory memory)
        {
            if (memory == null)
            {
                _links = IntPtr.Zero;
                _header = _links;
                _unusedLinksListMethods = null;
                _targetsTreeMethods = null;
                _unusedLinksListMethods = null;
            }
            else
            {
                _links = memory.Pointer;
                _header = _links;
                _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
                _targetsTreeMethods = new LinksTargetsTreeMethods(this);
                _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
            }
        }

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

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

        #region DisposableBase

        protected override bool AllowMultipleDisposeCalls => true;

        protected override void DisposeCore(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                SetPointers(null);
            }
            Disposable.TryDispose(_memory);
        }

        #endregion
    }
}
```

./ResizableDirectMemory/ResizableDirectMemoryLinks.ListMethods.cs

```
\texttt{
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);
        }
    }
}
}
```

./ResizableDirectMemory/ResizableDirectMemoryLinks.TreeMethods.cs

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

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    partial class ResizableDirectMemoryLinks<TLink>
    {
        private abstract class LinksTreeMethodsBase :
         ↪   SizedAndThreadedAVLBalancedTreeMethods<TLink>
        {
            private readonly ResizableDirectMemoryLinks<TLink> _memory;
            private readonly LinksCombinedConstants<TLink, TLink, int> _constants;
            protected readonly IntPtr Links;
            protected readonly IntPtr Header;

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

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

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

            public TLink this[TLink index]
            {
                get
                {
                    var root = GetTreeRoot();
                    if (GreaterOrEqualThan(index, GetSize(root)))
                    {
                        return GetZero();
                    }
                    while (!EqualToZero(root))
                    {
                        var left = GetLeftOrDefault(root);
                        var leftSize = GetSizeOrZero(left);
                        if (LessThan(index, leftSize))
                        {
                            root = left;
                            continue;
                        }
                        if (IsEquals(index, leftSize))
                        {
                            return root;
                        }
                        root = GetRightOrDefault(root);
                        index = Subtract(index, Increment(leftSize));
                    }
                    return GetZero(); // TODO: Impossible situation exception (only if tree
                     ↪   structure broken)
                }
            }

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

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

            public TLink EachReference(TLink link, Func<IList<TLink>, TLink> handler)
            {
                var root = GetTreeRoot();
                if (EqualToZero(root))
                {
```

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

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

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

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

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

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

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

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

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

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

        protected override void SetSize(TLink node, TLink size)
        {
            var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
            ↪   Link.SizeAsSourceOffset).GetValue<TLink>();
           (Links.GetElement(LinkSizeInBytes, node) + Link.SizeAsSourceOffset).SetValu⌋
            ↪   e(BitwiseHelpers.PartialWrite(previousValue, size, 5,
            ↪   -5));
        }

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

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

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

        protected override void SetRightIsChild(TLink node, bool value)
        {
            var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
            ↪   Link.SizeAsSourceOffset).GetValue<TLink>();
```

```
                var modified = BitwiseHelpers.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 3, 1);
                (Links.GetElement(LinkSizeInBytes, node) + Link.SizeAsSourceOffset).SetValue(modified);
            }

            protected override sbyte GetBalance(TLink node)
            {
                var previousValue = (Links.GetElement(LinkSizeInBytes, node) + Link.SizeAsSourceOffset).GetValue<TLink>();
                var value = (ulong)(Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 0, 3);
                var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 | 124 : value & 3);
                return unpackedValue;
            }

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

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

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

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

            protected override TLink GetBasePartValue(TLink link) => (Links.GetElement(LinkSizeInBytes, link) + Link.SourceOffset).GetValue<TLink>();

            /// <summary>
            /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target (концом)
            /// по дереву (индексу) связей, отсортированному по Source, а затем по Target.
            /// </summary>
            /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
            /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
            /// <returns>Индекс искомой связи.</returns>
            public TLink Search(TLink source, TLink target)
            {
                var root = GetTreeRoot();
                while (!EqualToZero(root))
                {
                    var rootSource = (Links.GetElement(LinkSizeInBytes, root) + Link.SourceOffset).GetValue<TLink>();
                    var rootTarget = (Links.GetElement(LinkSizeInBytes, root) + Link.TargetOffset).GetValue<TLink>();
                    if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) // node.Key < root.Key
                    {
                        root = GetLeftOrDefault(root);
                    }
                    else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) // node.Key > root.Key
                    {
                        root = GetRightOrDefault(root);
                    }
                    else // node.Key == root.Key
                    {
                        return root;
                    }
                }
                return GetZero();
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget, TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) || (IsEquals(firstSource, secondSource) && LessThan(firstTarget, secondTarget));

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget, TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) || (IsEquals(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
        }

        private class LinksTargetsTreeMethods : LinksTreeMethodsBase
        {
            public LinksTargetsTreeMethods(ResizableDirectMemoryLinks<TLink> memory)
                : base(memory)
            {
            }
        protected override IntPtr GetLeftPointer(TLink node) => Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsTargetOffset;

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

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

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

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

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

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

        protected override void SetSize(TLink node, TLink size)
        {
            var previousValue = (Links.GetElement(LinkSizeInBytes, node) + Link.SizeAsTargetOffset).GetValue<TLink>();
            (Links.GetElement(LinkSizeInBytes, node) + Link.SizeAsTargetOffset).SetValue(BitwiseHelpers.PartialWrite(previousValue, size, 5, -5));
        }

        protected override bool GetLeftIsChild(TLink node)
        {
            var previousValue = (Links.GetElement(LinkSizeInBytes, node) + Link.SizeAsTargetOffset).GetValue<TLink>();
            return (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 4, 1);
        }

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

        protected override bool GetRightIsChild(TLink node)
        {
            var previousValue = (Links.GetElement(LinkSizeInBytes, node) + Link.SizeAsTargetOffset).GetValue<TLink>();
            return (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 3, 1);
        }

        protected override void SetRightIsChild(TLink node, bool value)
        {
            var previousValue = (Links.GetElement(LinkSizeInBytes, node) + Link.SizeAsTargetOffset).GetValue<TLink>();
            var modified = BitwiseHelpers.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 3, 1);
            (Links.GetElement(LinkSizeInBytes, node) + Link.SizeAsTargetOffset).SetValue(modified);
        }

        protected override sbyte GetBalance(TLink node)
        {
            var previousValue = (Links.GetElement(LinkSizeInBytes, node) + Link.SizeAsTargetOffset).GetValue<TLink>();
            var value = (ulong)(Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 0, 3);
            var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 | 124 : value & 3);
            return unpackedValue;
        }

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

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

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

```

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

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

./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.cs

```
\texttt{
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Disposables;
using Platform.Collections.Arrays;
using Platform.Helpers.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) { }

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

        public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory, DefaultLinksSizeStep) { }

        public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long memoryReservationStep)
        {
            Constants = Default<LinksCombinedConstants<id, id, int>>.Instance;
            _memory = memory;
            _memoryReservationStep = memoryReservationStep;
            if (memory.ReservedCapacity < memoryReservationStep)
            {
                memory.ReservedCapacity = memoryReservationStep;
            }
            SetPointers(_memory);
            // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
            _memory.UsedCapacity = ((long)_header->AllocatedLinks * sizeof(Link)) + sizeof(LinksHeader);
            // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
            _header->ReservedLinks = (id)((_memory.ReservedCapacity -
            ↪   sizeof(LinksHeader)) / sizeof(Link));
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public id Count(IList<id> restrictions)
        {
            // Если нет ограничений, тогда возвращаем общее число связей находящихся в
            ↪   хранилище.
            if (restrictions.Count == 0)
            {
                return Total;
            }
            if (restrictions.Count == 1)
            {
                var index = restrictions[Constants.IndexPart];
                if (index == Constants.Any)
                {
                    return Total;
                }
                return Exists(index) ? 1UL : 0UL;
            }
            if (restrictions.Count == 2)
            {
                var index = restrictions[Constants.IndexPart];
                var value = restrictions[1];
                if (index == Constants.Any)
                {
                    if (value == Constants.Any)
                    {
                        return Total; // Any - как отсутствие ограничения
                    }
                    return _sourcesTreeMethods.CalculateReferences(value)
                         + _targetsTreeMethods.CalculateReferences(value);
                }
                else
                {
                    if (!Exists(index))
                    {
                        return 0;
                    }
                    if (value == Constants.Any)
                    {
                        return 1;
                    }
                    var storedLinkValue = GetLinkUnsafe(index);
                    if (storedLinkValue->Source == value ||
                        storedLinkValue->Target == value)
                    {
                        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.CalculateReferences(target);
                    }
                    else if (target == Constants.Any)
                    {
                        return _sourcesTreeMethods.CalculateReferences(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)
                    {
                        var link = _sourcesTreeMethods.Search(source, target);
                        return link == Constants.Null ? Constants.Continue : handler(GetLinkStruct(link));
                    }
                }
                else
                {
                    if (!Exists(index))
                    {
                        return Constants.Continue;
                    }
                    if (source == Constants.Any && target == Constants.Any)
                    {
                        return handler(GetLinkStruct(index));
                    }
                    var storedLinkValue = GetLinkUnsafe(index);
                    if (source != Constants.Any && target != Constants.Any)
                    {
                        if (storedLinkValue->Source == source &&
                            storedLinkValue->Target == target)
                        {
                            return handler(GetLinkStruct(index));
                        }
                        return Constants.Continue;
                    }
                    var value = default(id);
                    if (source == Constants.Any)
                    {
                        value = target;
                    }
                    if (target == Constants.Any)
                    {
                        value = source;
                    }
                    if (storedLinkValue->Source == value ||
                        storedLinkValue->Target == value)
                    {
                        return handler(GetLinkStruct(index));
                    }
                    return Constants.Continue;
                }
            }
            throw new NotSupportedException("Другие размеры и способы ограничений не поддерживаются.");
        }

        /// <remarks>
        /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует в другом месте (но не в менеджере памяти, а в логике Links)
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public id Update(IList<id> values)
        {
            var linkIndex = values[Constants.IndexPart];
            var link = GetLinkUnsafe(linkIndex);
            // Будет корректно работать только в том случае, если пространство выделенной связи предварительно заполнено нулями
            if (link->Source != Constants.Null)
            {
                _sourcesTreeMethods.Detach(new IntPtr(&_header->FirstAsSource), linkIndex);
            }
            if (link->Target != Constants.Null)
            {
                _targetsTreeMethods.Detach(new IntPtr(&_header->FirstAsTarget), linkIndex);
            }
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
            var leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
            var rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
            if (leftTreeSize != rightTreeSize)
            {
                throw new Exception("One of the trees is broken.");
            }
#endif
            link->Source = values[Constants.SourcePart];
            link->Target = values[Constants.TargetPart];
            if (link->Source != Constants.Null)
            {
                _sourcesTreeMethods.Attach(new IntPtr(&_header->FirstAsSource), linkIndex);
            }
            if (link->Target != Constants.Null)
            {
                _targetsTreeMethods.Attach(new IntPtr(&_header->FirstAsTarget), linkIndex);
            }
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
            leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
            rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
            if (leftTreeSize != rightTreeSize)
            {
                throw new Exception("One of the trees is broken.");
            }
#endif
            return linkIndex;
        }

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

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

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

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

        /// <remarks>
        /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том
        ↪   случае, если адрес реально поменялся
        ///
        /// Указатель this.links может быть в том же месте,
        /// так как 0-я связь не используется и имеет такой же размер как Header,
        /// поэтому header размещается в том же месте, что и 0-я связь
        /// </remarks>
        private void SetPointers(IResizableDirectMemory memory)
        {
            if (memory == null)
            {
                _header = null;
                _links = null;
                _unusedLinksListMethods = null;
                _targetsTreeMethods = null;
                _unusedLinksListMethods = null;
            }
            else
            {
                _header = (LinksHeader*)(void*)memory.Pointer;
                _links = (Link*)(void*)memory.Pointer;
                _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
                _targetsTreeMethods = new LinksTargetsTreeMethods(this);
                _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
            }
        }

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

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

        #region Disposable

        protected override bool AllowMultipleDisposeCalls => true;

        protected override void DisposeCore(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                SetPointers(null);
            }
            Disposable.TryDispose(_memory);
        }

        #endregion
    }
}
```

./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.ListMethods.cs

```
\texttt{
using Platform.Collections.Methods.Lists;

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

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

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

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

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

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

            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;
        }
    }
}
```

./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.TreeMethods.cs

```
\texttt{
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)
            }
        }

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

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

        protected override void PrintNodeValue(ulong node, StringBuilder sb)
        {
            sb.Append(' ');
            sb.Append(Links[node].Source);
            sb.Append('-');
```

```
                sb.Append('>');
                sb.Append(Links[node].Target);
            }
        }

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

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

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

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

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

            protected override ulong GetSize(ulong node)
            {
                var previousValue = Links[node].SizeAsSource;
                //return MathHelpers.PartialRead(previousValue, 5, -5);
                return (previousValue & 4294967264) >> 5;
            }

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

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

            protected override void SetSize(ulong node, ulong size)
            {
                var previousValue = Links[node].SizeAsSource;
                //var modified = MathHelpers.PartialWrite(previousValue, size, 5, -5);
                var modified = (previousValue & 31) | ((size & 134217727) << 5);
                Links[node].SizeAsSource = modified;
            }

            protected override bool GetLeftIsChild(ulong node)
            {
                var previousValue = Links[node].SizeAsSource;
                //return (Integer)MathHelpers.PartialRead(previousValue, 4, 1);
                return (previousValue & 16) >> 4 == 1UL;
            }

            protected override void SetLeftIsChild(ulong node, bool value)
            {
                var previousValue = Links[node].SizeAsSource;
                //var modified = MathHelpers.PartialWrite(previousValue,
                ↪   (ulong)(Integer)value, 4, 1);
                var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
                Links[node].SizeAsSource = modified;
            }

            protected override bool GetRightIsChild(ulong node)
            {
                var previousValue = Links[node].SizeAsSource;
                //return (Integer)MathHelpers.PartialRead(previousValue, 3, 1);
                return (previousValue & 8) >> 3 == 1UL;
            }

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

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

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

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

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

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

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

            /// <summary>
            /// Выполняет поиск и возвращает индекс связи с указанными Source
            ↪   (началом) и Target (концом)
            /// по дереву (индексу) связей, отсортированному по Source, а затем по Target.
            /// </summary>
            /// <param name="source">Индекс связи, которая является началом на
            ↪   искомой связи.</param>
            /// <param name="target">Индекс связи, которая является концом на искомой
            ↪   связи.</param>
            /// <returns>Индекс искомой связи.</returns>
            public ulong Search(ulong source, ulong target)
            {
                var root = Header->FirstAsSource;
                while (root != 0)
                {
                    var rootSource = Links[root].Source;
                    var rootTarget = Links[root].Target;
                if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) // node.Key < root.Key
                {
                    root = GetLeftOrDefault(root);
                }
                else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) // node.Key > root.Key
                {
                    root = GetRightOrDefault(root);
                }
                else // node.Key == root.Key
                {
                    return root;
                }
            }
            return 0;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget, ulong secondSource, ulong secondTarget)
            => firstSource < secondSource || (firstSource == secondSource && firstTarget < secondTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget, ulong secondSource, ulong secondTarget)
            => firstSource > secondSource || (firstSource == secondSource && firstTarget > secondTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(ulong node)
        {
            Links[node].LeftAsSource = 0UL;
            Links[node].RightAsSource = 0UL;
            Links[node].SizeAsSource = 0UL;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetZero() => 0UL;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetOne() => 1UL;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetTwo() => 2UL;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool ValueEqualToZero(IntPtr pointer) => *(ulong*)pointer.ToPointer() == 0UL;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool EqualToZero(ulong value) => value == 0UL;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool IsEquals(ulong first, ulong second) => first == second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThanZero(ulong value) => value > 0UL;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThan(ulong first, ulong second) => first > second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is always true for ulong

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessOrEqualThanZero(ulong value) => value == 0; // value is always >= 0 for ulong

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessThanZero(ulong value) => false; // value < 0 is always false for ulong

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessThan(ulong first, ulong second) => first < second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Increment(ulong value) => ++value;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Decrement(ulong value) => --value;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Add(ulong first, ulong second) => first + second;

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

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

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

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

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

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

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

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

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

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

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

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

            protected override ulong GetSize(ulong node)
            {
                var previousValue = Links[node].SizeAsTarget;
                //return MathHelpers.PartialRead(previousValue, 5, -5);
                return (previousValue & 4294967264) >> 5;
            }

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

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

            protected override void SetSize(ulong node, ulong size)
            {
                var previousValue = Links[node].SizeAsTarget;
                //var modified = MathHelpers.PartialWrite(previousValue, size, 5, -5);
                var modified = (previousValue & 31) | ((size & 134217727) << 5);
                Links[node].SizeAsTarget = modified;
            }

            protected override bool GetLeftIsChild(ulong node)
            {
                var previousValue = Links[node].SizeAsTarget;
                //return (Integer)MathHelpers.PartialRead(previousValue, 4, 1);
                return (previousValue & 16) >> 4 == 1UL;
                // TODO: Check if this is possible to use
                //var nodeSize = GetSize(node);
                //var left = GetLeftValue(node);
                //var leftSize = GetSizeOrZero(left);
                //return leftSize > 0 && nodeSize > leftSize;
            }

            protected override void SetLeftIsChild(ulong node, bool value)
            {
                var previousValue = Links[node].SizeAsTarget;
                //var modified = MathHelpers.PartialWrite(previousValue,
                ↪   (ulong)(Integer)value, 4, 1);
                var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
                Links[node].SizeAsTarget = modified;
            }

            protected override bool GetRightIsChild(ulong node)
            {
                var previousValue = Links[node].SizeAsTarget;
                //return (Integer)MathHelpers.PartialRead(previousValue, 3, 1);
                return (previousValue & 8) >> 3 == 1UL;
                // TODO: Check if this is possible to use
                //var nodeSize = GetSize(node);
                //var right = GetRightValue(node);
                //var rightSize = GetSizeOrZero(right);
                //return rightSize > 0 && nodeSize > rightSize;
            }

            protected override void SetRightIsChild(ulong node, bool value)
            {
                var previousValue = Links[node].SizeAsTarget;
                //var modified = MathHelpers.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 = MathHelpers.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 = MathHelpers.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;
            }
        }
    }
}
}
```

./Sequences/Converters/BalancedVariantConverter.cs

```
\texttt{
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];
            }
        }
    }
}
```

./Sequences/Converters/CompressingConverter.cs

```
\texttt{
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Collections;
using Platform.Helpers.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)
            {
                var newLength = ReplaceDoublets(copy);
                sequence = new TLink[newLength];
                for (int i = 0; i < newLength; i++)
                {
                    sequence[i] = copy[i].Element;
                }
            }
            return sequence;
        }

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void UpdateMaxDoublet(ref Doublet<TLink> doublet,
        ↪   LinkFrequency<TLink> data)
        {
            var frequency = data.Frequency;
            var maxFrequency = _maxDoubletData.Frequency;
            //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
            ↪   (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives
            ↪   better compression string data (and gives collisions quickly) */
            ↪   _maxDoublet.Source + _maxDoublet.Target)))
            if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
               (_comparer.Compare(maxFrequency, frequency) < 0 ||
                   (_equalityComparer.Equals(maxFrequency, frequency) &&
               ↪   _comparer.Compare(ArithmeticHelpers.Add(doublet.Source,
               ↪   doublet.Target), ArithmeticHelpers.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) */
            {
                _maxDoublet = doublet;
                _maxDoubletData = data;
            }
        }
    }
}
```

./Sequences/Converters/LinksListToSequenceConverterBase.cs

```
\texttt{
using System.Collections.Generic;
using Platform.Interfaces;

namespace Platform.Data.Doublets.Sequences.Converters
{
    public abstract class LinksListToSequenceConverterBase<TLink> :
    ↪   IConverter<IList<TLink>, TLink>
    {
        protected readonly ILinks<TLink> Links;
        public LinksListToSequenceConverterBase(ILinks<TLink> links) => Links = links;
        public abstract TLink Convert(IList<TLink> source);
    }
}
```

./Sequences/Converters/OptimalVariantConverter.cs

```
\texttt{
using System.Collections.Generic;
using System.Linq;
using Platform.Interfaces;

namespace Platform.Data.Doublets.Sequences.Converters
{
    public class OptimalVariantConverter<TLink> :
    ↪   LinksListToSequenceConverterBase<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪   EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        private readonly IConverter<IList<TLink>>
        ↪   _sequenceToItsLocalElementLevelsConverter;

        public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
        ↪   sequenceToItsLocalElementLevelsConverter) : base(links)
            => _sequenceToItsLocalElementLevelsConverter =
            ↪   sequenceToItsLocalElementLevelsConverter;

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

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

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

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

./Sequences/Converters/SequenceToItsLocalElementLevelsConverter.cs

```
\texttt{
using System.Collections.Generic;
using Platform.Interfaces;

namespace Platform.Data.Doublets.Sequences.Converters
{
    public class SequenceToItsLocalElementLevelsConverter<TLink> :
    ↪   LinksOperatorBase<TLink>, IConverter<IList<TLink>>
    {
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
        private readonly IConverter<Doublet<TLink>, TLink>
        ↪   _linkToItsFrequencyToNumberConveter;
        public SequenceToItsLocalElementLevelsConverter(ILinks<TLink> links,
        ↪   IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter)
        ↪   : base(links) => _linkToItsFrequencyToNumberConveter =
        ↪   linkToItsFrequencyToNumberConveter;
        public IList<TLink> Convert(IList<TLink> sequence)
        {
            var levels = new TLink[sequence.Count];
            levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
            for (var i = 1; i < sequence.Count - 1; i++)
            {
                var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
                var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
                levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
            }
            levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
            ↪   sequence[sequence.Count - 1]);
            return levels;
        }

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

./Sequences/CreteriaMatchers/DefaultSequenceElementCreteriaMatcher.cs

```
\texttt{
using Platform.Interfaces;

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

./Sequences/CreteriaMatchers/MarkedSequenceCreteriaMatcher.cs

```
\texttt{
using System.Collections.Generic;
using Platform.Interfaces;

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

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

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

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

./Sequences/DefaultSequenceAppender.cs

```
\texttt{
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);
        }
    }
}
}
```

./Sequences/DuplicateSegmentsCounter.cs

```
\texttt{
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);
    }
}
}
```

./Sequences/DuplicateSegmentsProvider.cs

```
\texttt{
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.Helpers;
using Platform.Helpers.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>>>>
```

```csharp
    {
        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) => HashHelpers.Generate(_listComparer.GetHashCode(pair.Key), _listComparer.GetHashCode(pair.Value));
        }

        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((HashSet<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();
        }
    }
}
```

./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkFrequencyIncrementer.cs

```
\texttt{
using System.Collections.Generic;
using Platform.Interfaces;

namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
{
    public class FrequenciesCacheBasedLinkFrequencyIncrementer<TLink> : IIncrementer<IList<TLink>>
    {
        private readonly LinkFrequenciesCache<TLink> _cache;

        public FrequenciesCacheBasedLinkFrequencyIncrementer(LinkFrequenciesCache<TLink> cache) => _cache = cache;

        /// <remarks>Sequence itseft is not changed, only frequency of its doublets is incremented.</remarks>
        public IList<TLink> Increment(IList<TLink> sequence)
        {
            _cache.IncrementFrequencies(sequence);
            return sequence;
        }
    }
}
```

./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkToItsFrequencyNumberConver

```
\texttt{
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;
    }
}
}
```

./Sequences/Frequencies/Cache/LinkFrequenciesCache.cs

```
\texttt{
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 = ArithmeticHelpers.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(ArithmeticHelpers.Subtract(frequency, count), Integer<TLink>.One) > 0))
                     || ((_comparer.Compare(count, frequency) > 0) && (_comparer.Compare(ArithmeticHelpers.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.");
                //    }
                //}
            }
        }
    }
}
}
```

./Sequences/Frequencies/Cache/LinkFrequency.cs

```
\texttt{
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 = ArithmeticHelpers<TLink>.Increment(Frequency);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void DecrementFrequency() => Frequency = ArithmeticHelpers<TLink>.Decrement(Frequency);
        public override string ToString() => $"F: {Frequency}, L: {Link}";
    }
}
}
```

./Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs

```
\texttt{
using Platform.Interfaces;

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

        public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
        ↪   ICreteriaMatcher<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();
        }
    }
}
}
```

./Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs

```
\texttt{
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;

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

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

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

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

./Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs

```
\texttt{
using Platform.Interfaces;

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

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

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

./Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.

```
\texttt{
using Platform.Interfaces;
using Platform.Numbers;

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    public class TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
    ↪   TotalSequenceSymbolFrequencyOneOffCounter<TLink>
    {
        private readonly ICreteriaMatcher<TLink> _markedSequenceMatcher;

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

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

./Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs

```
\texttt{
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();
    }
}
}
```

./Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs

```
\texttt{
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 = ArithmeticHelpers.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;
        }
    }
}
}
```

./Sequences/HeightProviders/CachedSequenceHeightProvider.cs

```
\texttt{
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 IPropertyOperator<TLink, TLink, TLink> _propertyOperator;

        public CachedSequenceHeightProvider(
            ILinks<TLink> links,
            ISequenceHeightProvider<TLink> baseHeightProvider,
            IConverter<TLink> addressToUnaryNumberConverter,
            IConverter<TLink> unaryNumberToAddressConverter,
            TLink heightPropertyMarker,
            IPropertyOperator<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;
        }
    }
}
```

./Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs

```
\texttt{
using Platform.Interfaces;
using Platform.Numbers;

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

        public DefaultSequenceRightHeightProvider(ILinks<TLink> links,
        ↪   ICreteriaMatcher<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 = ArithmeticHelpers.Increment(height);
            }
            return height;
        }
    }
}
```

./Sequences/HeightProviders/ISequenceHeightProvider.cs

```
\texttt{
using Platform.Interfaces;

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

./Sequences/Sequences.cs

```
\texttt{
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.Helpers.Singletons;
using LinkIndex = System.UInt64;
using Platform.Data.Constants;
using Platform.Data.Sequences;
using Platform.Data.Doublets.Sequences.Walkers;

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;

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

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

            Options.ValidateOptions();
            Options.InitOptions(Links);
        }

        public bool IsSequence(ulong sequence)
        {
            return Sync.ExecuteReadOperation(() =>
            {
                if (Options.UseSequenceMarker)
                {
                    return Options.MarkedSequenceMatcher.IsMatched(sequence);
                }
                return !Links.Unsync.IsPartialPoint(sequence);
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private ulong GetSequenceByElements(ulong sequence)
        {
            if (Options.UseSequenceMarker)
            {
                return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
            }
            return sequence;
        }

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

        #region Count

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

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

        #endregion

        #region Each

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

        public bool Each(Func<ulong, bool> handler, IList<ulong> sequence)
        {
            return Sync.ExecuteReadOperation(() =>
            {
                if (sequence.IsNullOrEmpty())
                {
                    return true;
                }
                Links.EnsureEachLinkIsAnyOrExists(sequence);
                if (sequence.Count == 1)
                {
                    var link = sequence[0];
                    if (link == _constants.Any)
                    {
                        return Links.Unsync.Each(_constants.Any, _constants.Any, handler);
                    }
                    return handler(link);
                }
                if (sequence.Count == 2)
                {
                    return Links.Unsync.Each(sequence[0], sequence[1], handler);
                }
                if (Options.UseIndex && !Options.Indexer.CheckIndex(sequence))
                {
                    return false;
                }
                return EachCore(handler, sequence);
            });
        }

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

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

        private bool StepRight(Func<ulong, bool> handler, ulong left, ulong right) => Links.Unsync.Each(left, _constants.Any, rightStep => TryStepRightUp(handler, right, rightStep));

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

        private bool StepLeft(Func<ulong, bool> handler, ulong left, ulong right) => Links.Unsync.Each(_constants.Any, right, leftStep => TryStepLeftUp(handler, left, leftStep));

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

        #endregion

        #region Update

        public ulong Update(ulong[] sequence, ulong[] newSequence)
        {
            if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
            {
                return _constants.Null;
            }
            if (sequence.IsNullOrEmpty())
            {
                return Create(newSequence);
            }
            if (newSequence.IsNullOrEmpty())
            {
                Delete(sequence);
                return _constants.Null;
            }
            return Sync.ExecuteWriteOperation(() =>
            {
                Links.EnsureEachLinkIsAnyOrExists(sequence);
                Links.EnsureEachLinkExists(newSequence);
                return UpdateCore(sequence, newSequence);
            });
        }

        private ulong UpdateCore(ulong[] sequence, ulong[] newSequence)
        {
            ulong bestVariant;
            if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew && !sequence.EqualTo(newSequence))
            {
                bestVariant = CompactCore(newSequence);
            }
            else
            {
                bestVariant = CreateCore(newSequence);
            }
            // TODO: Check all options only ones before loop execution
            // Возможно нужно две версии Each, возвращающий фактические последовательности и с маркером,
            // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты можно получить имея только фактические последовательности.
            foreach (var variant in Each(sequence))
            {
                if (variant != bestVariant)
                {
                    UpdateOneCore(variant, bestVariant);
                }
            }
            return bestVariant;
        }

        private void UpdateOneCore(ulong sequence, ulong newSequence)
        {
            if (Options.UseGarbageCollection)
            {
                var sequenceElements = GetSequenceElements(sequence);
                var sequenceElementsContents = new
                ↪   UInt64Link(Links.GetLink(sequenceElements));
                var sequenceLink = GetSequenceByElements(sequenceElements);
                var newSequenceElements = GetSequenceElements(newSequence);
                var newSequenceLink = GetSequenceByElements(newSequenceElements);
                if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
                {
                    if (sequenceLink != _constants.Null)
                    {
                        Links.Unsync.Merge(sequenceLink, newSequenceLink);
                    }
                    Links.Unsync.Merge(sequenceElements, newSequenceElements);
                }
                ClearGarbage(sequenceElementsContents.Source);
                ClearGarbage(sequenceElementsContents.Target);
            }
            else
            {
                if (Options.UseSequenceMarker)
                {
                    var sequenceElements = GetSequenceElements(sequence);
                    var sequenceLink = GetSequenceByElements(sequenceElements);
                    var newSequenceElements = GetSequenceElements(newSequence);
                    var newSequenceLink = GetSequenceByElements(newSequenceElements);
                    if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
                    {
                        if (sequenceLink != _constants.Null)
                        {
                            Links.Unsync.Merge(sequenceLink, newSequenceLink);
                        }
                        Links.Unsync.Merge(sequenceElements, newSequenceElements);
                    }
                }
                else
                {
                    if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
                    {
                        Links.Unsync.Merge(sequence, newSequence);
                    }
                }
            }
        }

        #endregion

        #region Delete

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

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

        #endregion

        #region Compactification

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

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

        #endregion

        #region Garbage Collection

        /// <remarks>
        /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted
        ↪   которое можно определить извне или в унаследованном классе
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool IsGarbage(ulong link) => link != Options.SequenceMarkerLink &&
        ↪   !Links.Unsync.IsPartialPoint(link) && Links.Count(link) == 0;

        private void ClearGarbage(ulong link)
        {
            if (IsGarbage(link))
            {
                var contents = new UInt64Link(Links.GetLink(link));
                Links.Unsync.Delete(link);
                ClearGarbage(contents.Source);
                ClearGarbage(contents.Target);
            }
        }

        #endregion

        #region Walkers

        public bool EachPart(Func<ulong, bool> handler, ulong sequence)
        {
            return Sync.ExecuteReadOperation(() =>
            {
                var links = Links.Unsync;
                var walker = new RightSequenceWalker<ulong>(links);
                foreach (var part in walker.Walk(sequence))
                {
                    if (!handler(links.GetIndex(part)))
                    {
                        return false;
                    }
                }
                return true;
            });
        }

        public class Matcher : RightSequenceWalker<ulong>
        {
            private readonly Sequences _sequences;
            private readonly IList<LinkIndex> _patternSequence;
            private readonly HashSet<LinkIndex> _linksInSequence;
            private readonly HashSet<LinkIndex> _results;
            private readonly Func<ulong, bool> _stopableHandler;
            private readonly HashSet<ulong> _readAsElements;
            private int _filterPosition;

            public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
            ↪   HashSet<LinkIndex> results, Func<LinkIndex, bool> stopableHandler,
            ↪   HashSet<LinkIndex> readAsElements = null)
                : base(sequences.Links.Unsync)
            {
                _sequences = sequences;
                _patternSequence = patternSequence;
                _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x
                ↪   != _constants.Any && x != ZeroOrMany));
                _results = results;
                _stopableHandler = stopableHandler;
                _readAsElements = readAsElements;
            }

            protected override bool IsElement(IList<ulong> link) => base.IsElement(link) ||
                (_readAsElements != null &&
            ↪   _readAsElements.Contains(Links.GetIndex(link))) ||
            ↪   _linksInSequence.Contains(Links.GetIndex(link));

            public bool FullMatch(LinkIndex sequenceToMatch)
            {
                _filterPosition = 0;
                foreach (var part in Walk(sequenceToMatch))
                {
                    if (!FullMatchCore(Links.GetIndex(part)))
                    {
                        break;
                    }
                }
                return _filterPosition == _patternSequence.Count;
            }

            private bool FullMatchCore(LinkIndex element)
            {
                if (_filterPosition == _patternSequence.Count)
                {
                    _filterPosition = -2; // Длиннее чем нужно
                    return false;
                }
                if (_patternSequence[_filterPosition] != _constants.Any
                 && element != _patternSequence[_filterPosition])
                {
                    _filterPosition = -1;
                    return false; // Начинается/Продолжается иначе
                }
                _filterPosition++;
                return true;
            }

            public void AddFullMatchedToResults(ulong sequenceToMatch)
            {
                if (FullMatch(sequenceToMatch))
                {
                    _results.Add(sequenceToMatch);
                }
            }

            public bool HandleFullMatched(ulong sequenceToMatch)
            {
                if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
                {
                    return _stopableHandler(sequenceToMatch);
                }
                return true;
            }

            public bool HandleFullMatchedSequence(ulong sequenceToMatch)
            {
                var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
                if (sequence != _constants.Null && FullMatch(sequenceToMatch) &&
                ↪    _results.Add(sequenceToMatch))
                {
                    return _stopableHandler(sequence);
                }
                return true;
            }

            /// <remarks>
            /// TODO: Add support for LinksConstants.Any
            /// </remarks>
            public bool PartialMatch(LinkIndex sequenceToMatch)
            {
                _filterPosition = -1;
                foreach (var part in Walk(sequenceToMatch))
                {
                    if (!PartialMatchCore(Links.GetIndex(part)))
                    {
                        break;
                    }
                }
                return _filterPosition == _patternSequence.Count - 1;
            }

            private bool PartialMatchCore(LinkIndex element)
            {
                if (_filterPosition == (_patternSequence.Count - 1))
                {
                    return false; // Нашлось
                }
                if (_filterPosition >= 0)
                {
                    if (element == _patternSequence[_filterPosition + 1])
                    {
                        _filterPosition++;
                    }
                    else
                    {
                        _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
    }
}
```

./Sequences/Sequences.Experiments.cs

```
\texttt{
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.Numbers;
using Platform.Data.Exceptions;
using Platform.Data.Sequences;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Walkers;

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)MathHelpers.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)MathHelpers.Catalan(sequence.Length));
                return CreateAllVariants1Core(sequence, results);
            });
        }

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

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

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

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

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

        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;
                });
                // |_x      ... x_o
                //  |_o      |___|
                Links.Each(_constants.Any, sequence[0], doublet =>
                {
                    var match = Links.SearchOrDefault(doublet, sequence[1]);
                    if (match != 0)
                    {
                        handler(match);
                    }
                    return true;
                });
                //          ._x o_.
                //           |___|
                PartialStepRight(x => handler(x), sequence[0], sequence[1]);
            }
            else
            {
                // TODO: Implement other variants
                return;
            }
        }

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

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

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

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

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

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

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

        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;

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

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

        private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
        ↪   Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
        ↪   LinkIndex[] knownElements)
        {
            var linksInSequence = new HashSet<ulong>(knownElements);
            //var entered = new HashSet<ulong>();
            var sb = new StringBuilder();
            sb.Append('{');
            if (links.Exists(sequenceLink))
            {
```

```
            StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
                x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => // entered.AddAndReturnVoid, x => { }, entered.DoNotContains
                {
                    if (insertComma && sb.Length > 1)
                    {
                        sb.Append(',');
                    }
                    //if (entered.Contains(element))
                    //{
                    //    sb.Append('{');
                    //    elementToString(sb, element);
                    //    sb.Append('}');
                    //}
                    //else
                    elementToString(sb, element);
                    if (sb.Length < MaxSequenceFormatSize)
                    {
                        return true;
                    }
                    sb.Append(insertComma ? ", ..." : "...");
                    return false;
                });
        }
        sb.Append('}');
        return sb.ToString();
    }

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

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

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

    public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
    {
        return Sync.ExecuteReadOperation(() =>
        {
            if (sequence.Length > 0)
            {
                Links.EnsureEachLinkExists(sequence);
                var results = new HashSet<ulong>();
                for (var i = 0; i < sequence.Length; i++)
                {
                    AllUsagesCore(sequence[i], results);
                }
                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(() =>
            {
                if (sequence.Length > 0)
                {
                    Links.EnsureEachLinkIsAnyOrExists(sequence);
                    var firstResults = new HashSet<ulong>();
                    var lastResults = new HashSet<ulong>();
                    var first = sequence.First(x => x != _constants.Any);
                    var last = sequence.Last(x => x != _constants.Any);
                    AllUsagesCore(first, firstResults);
                    AllUsagesCore(last, lastResults);
                    firstResults.IntersectWith(lastResults);
                    //for (var i = 0; i < sequence.Length; i++)
                    //    AllUsagesCore(sequence[i], results);
                    var filteredResults = new HashSet<ulong>();
                    var matcher = new Matcher(this, sequence, filteredResults, null);
                    matcher.AddAllPartialMatchedToResults(firstResults);
                    return filteredResults;
                }
                return new HashSet<ulong>();
            });
        }

        public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements, IList<ulong> sequence)
        {
            return Sync.ExecuteReadOperation(() =>
            {
                if (sequence.Count > 0)
                {
                    Links.EnsureEachLinkExists(sequence);
                    var results = new HashSet<LinkIndex>();
                    //var nextResults = new HashSet<ulong>();
                    //for (var i = 0; i < sequence.Length; i++)
                    //{
                    //    AllUsagesCore(sequence[i], nextResults);
                    //    if (results.IsNullOrEmpty())
                    //    {
                    //        results = nextResults;
                    //        nextResults = new HashSet<ulong>();
                    //    }
                    //    else
                    //    {
                    //        results.IntersectWith(nextResults);
                    //        nextResults.Clear();
                    //    }
                    //}
                    var collector1 = new AllUsagesCollector1(Links.Unsync, results);
                    collector1.Collect(Links.Unsync.GetLink(sequence[0]));
                    var next = new HashSet<ulong>();
                    for (var i = 1; i < sequence.Count; i++)
                    {
                        var collector = new AllUsagesCollector1(Links.Unsync, next);
                        collector.Collect(Links.Unsync.GetLink(sequence[i]));

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

        // Does not work
        public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements, params ulong[] sequence)
        {
            var visited = new HashSet<ulong>();
            var results = new HashSet<ulong>();
            var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return true; }, readAsElements);
            var last = sequence.Length - 1;
            for (var i = 0; i < last; i++)
            {
                PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
            }
            return results;
        }

        public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
        {
            return Sync.ExecuteReadOperation(() =>
            {
                if (sequence.Length > 0)
                {
                    Links.EnsureEachLinkExists(sequence);
                    //var firstElement = sequence[0];
                    //if (sequence.Length == 1)
                    //{
                    //    //results.Add(firstElement);
                    //    return results;
                    //}
                    //if (sequence.Length == 2)
                    //{
                    //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
                    //    //if (doublet != Doublets.Links.Null)
                    //    //    results.Add(doublet);
                    //    return results;
                    //}
                    //var lastElement = sequence[sequence.Length - 1];
                    //Func<ulong, bool> handler = x =>
                    //{
                    //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement)) results.Add(x);
                    //    return true;
                    //};
                    //if (sequence.Length >= 2)
                    //    StepRight(handler, sequence[0], sequence[1]);
                    //var last = sequence.Length - 2;
                    //for (var i = 1; i < last; i++)
                    //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
                    //if (sequence.Length >= 3)
                    //    StepLeft(handler, sequence[sequence.Length - 2], sequence[sequence.Length - 1]);
                    //////if (sequence.Length == 1)
                    //////{
                    //////    throw new NotImplementedException(); // all sequences, containing this element?
                    //////}
                    //////if (sequence.Length == 2)
                    //////{
                    //////    var results = new List<ulong>();
                    //////    PartialStepRight(results.Add, sequence[0], sequence[1]);
                    //////    return results;
                    //////}
                    //////var matches = new List<List<ulong>>();
                    //////var last = sequence.Length - 1;
                    //////for (var i = 0; i < last; i++)
                    //////{
                    //////    var results = new List<ulong>();
                    //////    //StepRight(results.Add, sequence[i], sequence[i + 1]);
                    //////    PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
                    //////    if (results.Count > 0)
                    //////        matches.Add(results);
                    //////    else
                    //////        return results;
                    //////    if (matches.Count == 2)
                    //////    {
                    //////        var merged = new List<ulong>();
                    //////        for (var j = 0; j < matches[0].Count; j++)
                    //////            for (var k = 0; k < matches[1].Count; k++)
                    //////                CloseInnerConnections(merged.Add, matches[0][j], matches[1][k]);
                    //////        if (merged.Count > 0)
                    //////            matches = new List<List<ulong>> { merged };
                //////        else
                //////            return new List<ulong>();
                //////    }
                //////}
                //////if (matches.Count > 0)
                //////{
                //////    var usages = new HashSet<ulong>();
                //////    for (int i = 0; i < sequence.Length; i++)
                //////    {
                //////        AllUsagesCore(sequence[i], usages);
                //////    }
                //////    //for (int i = 0; i < matches[0].Count; i++)
                //////    //    AllUsagesCore(matches[0][i], usages);
                //////    //usages.UnionWith(matches[0]);
                //////    return usages.ToList();
                //////}
                var firstLinkUsages = new HashSet<ulong>();
                AllUsagesCore(sequence[0], firstLinkUsages);
                firstLinkUsages.Add(sequence[0]);
                //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
                ↪   sequence[0] }; // or all sequences, containing this element?
                //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
                ↪   1).ToList();
                var results = new HashSet<ulong>();
                foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
                ↪   firstLinkUsages, 1))
                {
                    AllUsagesCore(match, results);
                }
                return results.ToList();
            }
            return new List<ulong>();
        });
    }

    /// <remarks>
    /// TODO: Может потробоваться ограничение на уровень глубины рекурсии
    /// </remarks>
    public HashSet<ulong> AllUsages(ulong link)
    {
        return Sync.ExecuteReadOperation(() =>
        {
            var usages = new HashSet<ulong>();
            AllUsagesCore(link, usages);
            return usages;
        });
    }

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

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

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

    public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
    {
        if (Options.UseSequenceMarker)
        {
            var counter = new
                ↪   TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
                ↪   Options.MarkedSequenceMatcher, symbol);
            return counter.Count();
        }
        else
        {
            var counter = new
                ↪   TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links, symbol);
            return counter.Count();
        }
    }

    private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<ulong, bool>
    ↪   outerHandler)
    {
        bool handler(ulong doublet)
        {
            if (usages.Add(doublet))
            {
                if (!outerHandler(doublet))
                {
                    return false;
                }
                if (!AllUsagesCore1(doublet, usages, outerHandler))
                {
                    return false;
                }
            }
            return true;
        }
        return Links.Unsync.Each(link, _constants.Any, handler)
            && Links.Unsync.Each(_constants.Any, link, handler);
    }

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

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

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

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

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

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

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

        public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
        {
            _links = links;
            _totals = totals;
        }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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) // ?
            {
                return previousMatchings;
            }
            var secondLinkUsages = new HashSet<ulong>();
            AllUsagesCore(sequence[startAt], secondLinkUsages);
            secondLinkUsages.Add(sequence[startAt]);
            var matchings = new HashSet<ulong>();
            //for (var i = 0; i < previousMatchings.Count; i++)
            foreach (var secondLinkUsage in secondLinkUsages)
            {
                foreach (var previousMatching in previousMatchings)
                {
                    //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching, secondLinkUsage);
                    StepRight(matchings.AddAndReturnVoid, previousMatching, secondLinkUsage);
                    TryStepRightUp(matchings.AddAndReturnVoid, secondLinkUsage, previousMatching);
                    //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage, sequence[startAt]); // почему-то эта ошибочная запись приводит к желаемым результатам.
                    PartialStepRight(matchings.AddAndReturnVoid, previousMatching, secondLinkUsage);
                }
            }
            if (matchings.Count == 0)
            {
                return matchings;
            }
            return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
        }

        private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong> links, params ulong[] sequence)
        {
            if (sequence == null)
            {
                return;
            }
            for (var i = 0; i < sequence.Length; i++)
            {
                if (sequence[i] != _constants.Any && sequence[i] != ZeroOrMany && !links.Exists(sequence[i]))
                {
                    throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i], $"patternSequence[{i}]");
                }
            }
        }

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

        // 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

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

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

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

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

./Sequences/Sequences.Experiments.ReadSequence.cs

```
\texttt{
//#define USEARRAYPOOL
using System;
using System.Runtime.CompilerServices;
#if USEARRAYPOOL
using Platform.Collections;
#endif

namespace Platform.Data.Doublets.Sequences
{
    partial class Sequences
    {
        public ulong[] ReadSequenceCore(ulong sequence, Func<ulong, bool> isElement)
        {
            var links = Links.Unsync;
            var length = 1;
            var array = new ulong[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 ulong[length];
#endif
                hasElements = false;
                for (var i = 0; i < array.Length; i++)
                {
                    var candidate = array[i];
                    if (candidate == 0)
                    {
                        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 ulong[] CopyFilledElements(ulong[] array, int filledElementsCount)
        {
            var finalArray = new ulong[filledElementsCount];
            for (int i = 0, j = 0; i < array.Length; i++)
            {
                if (array[i] > 0)
                {
                    finalArray[j] = array[i];
                    j++;
                }
            }
#if USEARRAYPOOL
                ArrayPool.Free(array);
#endif
            return finalArray;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static int CountFilledElements(ulong[] array)
        {
            var count = 0;
            for (var i = 0; i < array.Length; i++)
            {
                if (array[i] > 0)
                {
                    count++;
                }
            }
            return count;
        }
    }
}
```

./Sequences/SequencesExtensions.cs

```
\texttt{
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);
        }
    }
}
```

./Sequences/SequencesIndexer.cs

```
\texttt{
using System.Collections.Generic;

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

        private readonly ISynchronizedLinks<TLink> _links;
        private readonly TLink _null;

        public SequencesIndexer(ISynchronizedLinks<TLink> links)
        {
            _links = links;
            _null = _links.Constants.Null;
        }

        /// <summary>
        /// Индексирует последовательность глобально, и возвращает значение,
        /// определяющие была ли запрошенная последовательность проиндексирована
        ↪   ранее.
        /// </summary>
        /// <param name="sequence">Последовательность для индексации.</param>
        /// <returns>
        /// True если последовательность уже была проиндексирована ранее и
        /// False если последовательность была проиндексирована только что.
        /// </returns>
        public bool Index(TLink[] sequence)
        {
            var indexed = true;
            var i = sequence.Length;
            while (--i >= 1 && (indexed =
            ↪   !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1],
            ↪   sequence[i]), _null))) { }
            for (; i >= 1; i--)
            {
                _links.GetOrCreate(sequence[i - 1], sequence[i]);
            }
            return indexed;
        }

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

        public bool BulkIndexUnsync(TLink[] sequence)
        {
            var indexed = true;
            var i = sequence.Length;
            var links = _links.Unsync;
            while (--i >= 1 && (indexed = !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1], sequence[i]), _null))) { }
            for (; i >= 1; i--)
            {
                links.GetOrCreate(sequence[i - 1], sequence[i]);
            }
            return indexed;
        }

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

./Sequences/SequencesOptions.cs

```
\texttt{
using System;
using System.Collections.Generic;
using Platform.Interfaces;
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;

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 MarkedSequenceCreteriaMatcher<TLink> MarkedSequenceMatcher { get; set; }
        public IConverter<IList<TLink>, TLink> LinksToSequenceConverter { get; set; }
        public SequencesIndexer<TLink> Indexer { 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 MarkedSequenceCreteriaMatcher<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 && Indexer == null)
            {
                Indexer = new SequencesIndexer<TLink>(links);
            }
        }

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

./Sequences/UnicodeMap.cs

```
\texttt{
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(FromLinkT ⌋
                           ↪   oChar(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;
        }
    }
}
```

./Sequences/Walkers/LeftSequenceWalker.cs

```
\texttt{
using System.Collections.Generic;
using System.Runtime.CompilerServices;

namespace Platform.Data.Doublets.Sequences.Walkers
{
    public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
    {
        public LeftSequenceWalker(ILinks<TLink> links) : base(links) { }

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

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

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

./Sequences/Walkers/RightSequenceWalker.cs

```
\texttt{
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

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

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

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

./Sequences/Walkers/SequenceWalkerBase.cs

```
\texttt{
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Data.Sequences;

namespace Platform.Data.Doublets.Sequences.Walkers
{
    public abstract class SequenceWalkerBase<TLink> : LinksOperatorBase<TLink>,
    ↪   ISequenceWalker<TLink>
    {
        // TODO: Use IStack indead of System.Collections.Generic.Stack, but IStack should
        ↪   contain IsEmpty property
        private readonly Stack<IList<TLink>> _stack;

        protected SequenceWalkerBase(ILinks<TLink> links) : base(links) => _stack = new
        ↪   Stack<IList<TLink>>();

        public IEnumerable<IList<TLink>> Walk(TLink sequence)
        {
            if (_stack.Count > 0)
            {
                _stack.Clear(); // This can be replaced with while(!_stack.IsEmpty)
                ↪   _stack.Pop()
            }
            var element = Links.GetLink(sequence);
            if (IsElement(element))
            {
                yield return element;
            }
            else
            {
                while (true)
                {
                    if (IsElement(element))
                    {
                        if (_stack.Count == 0)
                        {
                            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(IList<TLink> elementLink) =>
        ↪   Point<TLink>.IsPartialPointUnchecked(elementLink);

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

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

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

./Stacks/Stack.cs

```
\texttt{
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 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));
    }
}
```

./Stacks/StackExtensions.cs

```
\texttt{
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;
        }

        public static void DeleteStack<TLink>(this ILinks<TLink> links, TLink stack) =>
        ↪   links.Delete(stack);
    }
}
```

./SynchronizedLinks.cs

```
\texttt{
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);
        //}
    }
}
```

./UInt64Link.cs

```
\texttt{
using System;
using System.Collections;
using System.Collections.Generic;
using Platform.Exceptions;
using Platform.Ranges;
using Platform.Helpers.Singletons;
using Platform.Data.Constants;

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

```csharp
    /// 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 ulong[] ToArray()
        {
            var array = new ulong[Length];
            CopyTo(array, 0);
            return array;
        }

        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;
        }

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

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

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

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

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

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

            return -1;
        }

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

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

        #endregion
    }
}
```

./UInt64LinkExtensions.cs

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

./UInt64LinksExtensions.cs

```
\texttt{
using System;
using System.Text;
using System.Collections.Generic;
using Platform.Helpers.Singletons;
using Platform.Data.Constants;
using Platform.Data.Exceptions;
using Platform.Data.Doublets.Sequences;

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

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

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

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

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

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

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

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

./UInt64LinksTransactionsLayer.cs

```
\texttt{
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>
        public struct Transition
        {
            public static readonly long Size = StructureHelpers.SizeOf<Transition>();

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

            public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId, UInt64Link before, UInt64Link after)
            {
                TransactionId = transactionId;
                Before = before;
                After = after;
                Timestamp = uniqueTimestampFactory.Create();
            }

            public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId, UInt64Link before)
                : this(uniqueTimestampFactory, transactionId, before, default)
            {
            }

            public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId)
                : this(uniqueTimestampFactory, transactionId, default, default)
            {
            }

            public override string ToString() => $"{Timestamp} {TransactionId}: {Before} => {After}";
        }

        /// <remarks>
        /// Другие варианты реализации транзакций (атомарности):
        ///     1. Разделение хранения значения связи ((Source Target) или (Source Linker Target)) и индексов.
        ///     2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно потребуется решить вопрос
        ///        со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с пересечениями идентификаторов.
        ///
        /// Где хранить промежуточный список транзакций?
        ///
        /// В оперативной памяти:
        ///  Минусы:
```

```
        ///     1. Может усложнить систему, если она будет функционировать
        ↪   самостоятельно,
        ///     так как нужно отдельно выделять память под список трансформаций.
        ///     2. Выделенной оперативной памяти может не хватить, в том случае,
        ///     если транзакция использует слишком много трансформаций.
        ///         -> Можно использовать жёсткий диск для слишком длинных транзакций.
        ///         -> Максимальный размер списка трансформаций можно ограничить /
        ↪   задать константой.
        ///     3. При подтверждении транзакции (Commit) все трансформации
        ↪   записываются разом создавая задержку.
        ///
        /// На жёстком диске:
        ///  Минусы:
        ///     1. Длительный отклик, на запись каждой трансформации.
        ///     2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
        ///         -> Это может решаться упаковкой/исключением дублирующих операций.
        ///         -> Также это может решаться тем, что короткие транзакции вообще
        ///             не будут записываться в случае отката.
        ///     3. Перед тем как выполнять отмену операций транзакции нужно дождаться
        ↪   пока все операции (трансформации)
        ///         будут записаны в лог.
        ///
        /// </remarks>
        public class Transaction : DisposableBase
        {
            private readonly Queue<Transition> _transitions;
            private readonly UInt64LinksTransactionsLayer _layer;
            public bool IsCommitted { get; private set; }
            public bool IsReverted { get; private set; }

            public Transaction(UInt64LinksTransactionsLayer layer)
            {
                _layer = layer;
                if (_layer._currentTransactionId != 0)
                {
                    throw new NotSupportedException("Nested transactions not supported.");
                }
                IsCommitted = false;
                IsReverted = false;
                _transitions = new Queue<Transition>();
                SetCurrentTransaction(layer, this);
            }

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

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

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

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

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

            // TODO: THIS IS EXCEPTION WORKAROUND, REMOVE IT THEN
            ↪   https://github.com/linksplatform/Disposables/issues/13 FIXED
            protected override bool AllowMultipleDisposeCalls => true;
        }

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

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

        public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
            : base(links)
        {
            if (string.IsNullOrWhiteSpace(logAddress))
            {
                throw new ArgumentNullException(nameof(logAddress));
            }
            // В первой строке файла хранится последняя закоммиченную транзакцию.
            // При запуске это используется для проверки удачного закрытия файла лога.
            // In the first line of the file the last committed transaction is stored.
            // On startup, this is used to check that the log file is successfully closed.
            var lastCommitedTransition =
            ↪   FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
            var lastWrittenTransition =
            ↪   FileHelpers.ReadLastOrDefault<Transition>(logAddress);
            if (!lastCommitedTransition.Equals(lastWrittenTransition))
            {
                Dispose();
                throw new NotSupportedException("Database is damaged, autorecovery is not
                ↪   supported yet.");
            }
            if (lastCommitedTransition.Equals(default(Transition)))
            {
                FileHelpers.WriteFirst(logAddress, lastCommitedTransition);
            }
            _lastCommitedTransition = lastCommitedTransition;
            // TODO: Think about a better way to calculate or store this value
            var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
            _lastCommitedTransactionId = allTransitions.Max(x => x.TransactionId);
            _uniqueTimestampFactory = new UniqueTimestampFactory();
            _logAddress = logAddress;
            _log = FileHelpers.Append(logAddress);
            _transitions = new Queue<Transition>();
            _transitionsPusher = new Task(TransitionsPusher);
            _transitionsPusher.Start();
        }

        public IList<ulong> GetLinkValue(ulong link) => Links.GetLink(link);

        public override ulong Create()
        {
            var createdLinkIndex = Links.Create();
            var createdLink = new UInt64Link(Links.GetLink(createdLinkIndex));
            CommitTransition(new Transition(_uniqueTimestampFactory,
            ↪   _currentTransactionId, default, createdLink));
            return createdLinkIndex;
        }

        public override ulong Update(IList<ulong> parts)
        {
            var beforeLink = new UInt64Link(Links.GetLink(parts[Constants.IndexPart]));
            parts[Constants.IndexPart] = Links.Update(parts);
            var afterLink = new UInt64Link(Links.GetLink(parts[Constants.IndexPart]));
            CommitTransition(new Transition(_uniqueTimestampFactory,
            ↪   _currentTransactionId, beforeLink, afterLink));
            return parts[Constants.IndexPart];
        }

        public override void Delete(ulong link)
        {
            var deletedLink = new UInt64Link(Links.GetLink(link));
            Links.Delete(link);
            CommitTransition(new Transition(_uniqueTimestampFactory,
            ↪   _currentTransactionId, deletedLink, default));
        }

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

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

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

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

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

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

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

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

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

        #region DisposalBase

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

        #endregion
    }
}
}
```

Index