./Converters/AddressToUnaryNumberConverter.cs

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

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

        private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;

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

        public TLink Convert(TLink sourceAddress)
        {
            var number = sourceAddress;
            var 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

```
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

```
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

```
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

```
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.Convert(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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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)
```

```
58              {
59                  throw new NotSupportedException();
60              }
61              var updatedLink = restrictions[Constants.IndexPart];
62              this.EnsureLinkExists(updatedLink, nameof(Constants.IndexPart));
63              var newSource = restrictions[Constants.SourcePart];
64              this.EnsureLinkIsItselfOrExists(newSource, nameof(Constants.SourcePart));
65              var newTarget = restrictions[Constants.TargetPart];
66              this.EnsureLinkIsItselfOrExists(newTarget, nameof(Constants.TargetPart));
67              var existedLink = Constants.Null;
68              if (newSource != Constants.Itself && newTarget != Constants.Itself)
69              {
70                  existedLink = this.SearchOrDefault(newSource, newTarget);
71              }
72              if (existedLink == Constants.Null)
73              {
74                  var before = Links.GetLink(updatedLink);
75                  if (before[Constants.SourcePart] != newSource || before[Constants.TargetPart] !=
                    ↪  newTarget)
76                  {
77                      Links.Update(updatedLink, newSource == Constants.Itself ? updatedLink :
                        ↪  newSource,
78                                                newTarget == Constants.Itself ? updatedLink :
                                                  ↪  newTarget);
79                  }
80                  return updatedLink;
81              }
82              else
83              {
84                  // Replace one link with another (replaced link is deleted, children are updated
                    ↪  or deleted), it is actually merge operation
85                  return this.Merge(updatedLink, existedLink);
86              }
87          }
88
89          /// <summary>Удаляет связь с указанным индексом.</summary>
90          /// <param name="link">Индекс удаляемой связи.</param>
91          public override void Delete(ulong link)
92          {
93              this.EnsureLinkExists(link);
94              Links.Update(link, Constants.Null, Constants.Null);
95              var referencesCount = Links.Count(Constants.Any, link);
96              if (referencesCount > 0)
97              {
98                  var references = new ulong[referencesCount];
99                  var referencesFiller = new ArrayFiller<ulong, ulong>(references,
                    ↪  Constants.Continue);
100                 Links.Each(referencesFiller.AddFirstAndReturnConstant, Constants.Any, link);
101                 //references.Sort(); // TODO: Решить необходимо ли для корректного порядка
                    ↪  отмены операций в транзакциях
102                 for (var i = (long)referencesCount - 1; i >= 0; i--)
103                 {
104                     if (this.Exists(references[i]))
105                     {
106                         Delete(references[i]);
107                     }
108                 }
109                 //else
110                 // TODO: Определить почему здесь есть связи, которых не существует
111             }
112             Links.Delete(link);
113         }
114     }
115 }
```

./Decorators/UniLinks.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using Platform.Collections;
5  using Platform.Collections.Arrays;
6  using Platform.Collections.Lists;
7  using Platform.Helpers.Scopes;
8  using Platform.Data.Constants;
9  using Platform.Data.Universal;
10 using System.Collections.ObjectModel;
11
12 namespace Platform.Data.Doublets.Decorators
13 {
```

```
14      /// <remarks>
15      /// What does empty pattern (for condition or substitution) mean? Nothing or Everything?
16      /// 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).
17      ///
18      /// 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.)
19      /// </remarks>
20      internal class UniLinks<TLink> : LinksDecoratorBase<TLink>, IUniLinks<TLink>
21      {
22          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
23
24          public UniLinks(ILinks<TLink> links) : base(links) { }
25
26          private struct Transition
27          {
28              public IList<TLink> Before;
29              public IList<TLink> After;
30
31              public Transition(IList<TLink> before, IList<TLink> after)
32              {
33                  Before = before;
34                  After = after;
35              }
36          }
37
38          public static readonly TLink NullConstant = Use<LinksCombinedConstants<TLink, TLink,
            ↪  int>>.Single.Null;
39          public static readonly IReadOnlyList<TLink> NullLink = new ReadOnlyCollection<TLink>(new
            ↪  List<TLink> { NullConstant, NullConstant, NullConstant });
40
41          // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution
            ↪  (Links-Expression)
42          public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>, TLink>
            ↪  matchedHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
            ↪  substitutedHandler)
43          {
44              ////List<Transition> transitions = null;
45              ////if (!restriction.IsNullOrEmpty())
46              ////{
47              ////    // Есть причина делать проход (чтение)
48              ////    if (matchedHandler != null)
49              ////    {
50              ////        if (!substitution.IsNullOrEmpty())
51              ////        {
52              ////            // restriction => { 0, 0, 0 } | { 0 } // Create
53              ////            // substitution => { itself, 0, 0 } | { itself, itself, itself } //
                ↪  Create / Update
54              ////            // substitution => { 0, 0, 0 } | { 0 } // Delete
55              ////            transitions = new List<Transition>();
56              ////            if (Equals(substitution[Constants.IndexPart], Constants.Null))
57              ////            {
58              ////                // If index is Null, that means we always ignore every other
                ↪  value (they are also Null by definition)
59              ////                var matchDecision = matchedHandler(, NullLink);
60              ////                if (Equals(matchDecision, Constants.Break))
61              ////                    return false;
62              ////                if (!Equals(matchDecision, Constants.Skip))
63              ////                    transitions.Add(new Transition(matchedLink, newValue));
64              ////            }
65              ////            else
66              ////            {
67              ////                Func<T, bool> handler;
68              ////                handler = link =>
69              ////                {
70              ////                    var matchedLink = Memory.GetLinkValue(link);
71              ////                    var newValue = Memory.GetLinkValue(link);
72              ////                    newValue[Constants.IndexPart] = Constants.Itself;
73              ////                    newValue[Constants.SourcePart] =
                ↪  Equals(substitution[Constants.SourcePart], Constants.Itself) ?
                ↪  matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
74              ////                    newValue[Constants.TargetPart] =
                ↪  Equals(substitution[Constants.TargetPart], Constants.Itself) ?
                ↪  matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
75              ////                    var matchDecision = matchedHandler(matchedLink, newValue);
76              ////                    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 =>
```

```
154             //        {
155             //            var matchedLink = Memory.GetLinkValue(link);
156             //            var matchDecision = matchedHandler(matchedLink);
157             //            if (Equals(matchDecision, Constants.Break))
158             //                return false;
159             //            if (!Equals(matchDecision, Constants.Skip))
160             //                matchedLinks.Add(matchedLink);
161             //            return true;
162             //        };
163             //        if (!Memory.Each(handler, restriction))
164             //            return Constants.Break;
165             //    }
166             //    if (!matchedLinks.IsNullOrEmpty())
167             //    {
168             //        var totalMatchedLinks = matchedLinks.Count;
169             //        for (var i = 0; i < totalMatchedLinks; i++)
170             //        {
171             //            var matchedLink = matchedLinks[i];
172             //            if (substitutedHandler != null)
173             //            {
174             //                var newValue = new List<T>(); // TODO: Prepare value to update here
175             //                // TODO: Decide is it actually needed to use Before and After
                ↪  substitution handling.
176             //                var substitutedDecision = substitutedHandler(matchedLink,
                ↪  newValue);
177             //                if (Equals(substitutedDecision, Constants.Break))
178             //                    return Constants.Break;
179             //                if (Equals(substitutedDecision, Constants.Continue))
180             //                {
181             //                    // Actual update here
182             //                    Memory.SetLinkValue(newValue);
183             //                }
184             //                if (Equals(substitutedDecision, Constants.Skip))
185             //                {
186             //                    // Cancel the update. TODO: decide use separate Cancel
                ↪  constant or Skip is enough?
187             //                }
188             //            }
189             //        }
190             //    }
191             //}
192             return Constants.Continue;
193         }
194
195         public TLink Trigger(IList<TLink> patternOrCondition, Func<IList<TLink>, TLink>
            ↪  matchHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
            ↪  substitutionHandler)
196         {
197             if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
198             {
199                 return Constants.Continue;
200             }
201             else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO:
                ↪  Check if it is a correct condition
202             {
203                 // Or it only applies to trigger without matchHandler.
204                 throw new NotImplementedException();
205             }
206             else if (!substitution.IsNullOrEmpty()) // Creation
207             {
208                 var before = ArrayPool<TLink>.Empty;
209                 // Что должно означать False здесь? Остановиться (перестать идти) или пропустить
                    ↪  (пройти мимо) или пустить (взять)?
210                 if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                    ↪  Constants.Break))
211                 {
212                     return Constants.Break;
213                 }
214                 var after = (IList<TLink>)substitution.ToArray();
215                 if (_equalityComparer.Equals(after[0], default))
216                 {
217                     var newLink = Links.Create();
218                     after[0] = newLink;
219                 }
220                 if (substitution.Count == 1)
221                 {
222                     after = Links.GetLink(substitution[0]);
223                 }
```

```
224                     else if (substitution.Count == 3)
225                     {
226                         Links.Update(after);
227                     }
228                     else
229                     {
230                         throw new NotSupportedException();
231                     }
232                     if (matchHandler != null)
233                     {
234                         return substitutionHandler(before, after);
235                     }
236                     return Constants.Continue;
237                 }
238                 else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
239                 {
240                     if (patternOrCondition.Count == 1)
241                     {
242                         var linkToDelete = patternOrCondition[0];
243                         var before = Links.GetLink(linkToDelete);
244                         if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                            ↪ Constants.Break))
245                         {
246                             return Constants.Break;
247                         }
248                         var after = ArrayPool<TLink>.Empty;
249                         Links.Update(linkToDelete, Constants.Null, Constants.Null);
250                         Links.Delete(linkToDelete);
251                         if (matchHandler != null)
252                         {
253                             return substitutionHandler(before, after);
254                         }
255                         return Constants.Continue;
256                     }
257                     else
258                     {
259                         throw new NotSupportedException();
260                     }
261                 }
262                 else // Replace / Update
263                 {
264                     if (patternOrCondition.Count == 1) //-V3125
265                     {
266                         var linkToUpdate = patternOrCondition[0];
267                         var before = Links.GetLink(linkToUpdate);
268                         if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                            ↪ Constants.Break))
269                         {
270                             return Constants.Break;
271                         }
272                         var after = (IList<TLink>)substitution.ToArray(); //-V3125
273                         if (_equalityComparer.Equals(after[0], default))
274                         {
275                             after[0] = linkToUpdate;
276                         }
277                         if (substitution.Count == 1)
278                         {
279                             if (!_equalityComparer.Equals(substitution[0], linkToUpdate))
280                             {
281                                 after = Links.GetLink(substitution[0]);
282                                 Links.Update(linkToUpdate, Constants.Null, Constants.Null);
283                                 Links.Delete(linkToUpdate);
284                             }
285                         }
286                         else if (substitution.Count == 3)
287                         {
288                             Links.Update(after);
289                         }
290                         else
291                         {
292                             throw new NotSupportedException();
293                         }
294                         if (matchHandler != null)
295                         {
296                             return substitutionHandler(before, after);
297                         }
298                         return Constants.Continue;
299                     }
```

```
                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

```
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

```
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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

```
65          public static explicit operator int(Hybrid<T> hybrid) =>
            ↪ Convert.ToInt32(hybrid.AbsoluteValue);
66
67          public static explicit operator ushort(Hybrid<T> hybrid) =>
            ↪ Convert.ToUInt16(hybrid.Value);
68
69          public static explicit operator short(Hybrid<T> hybrid) =>
            ↪ Convert.ToInt16(hybrid.AbsoluteValue);
70
71          public static explicit operator byte(Hybrid<T> hybrid) => Convert.ToByte(hybrid.Value);
72
73          public static explicit operator sbyte(Hybrid<T> hybrid) =>
            ↪ Convert.ToSByte(hybrid.AbsoluteValue);
74
75          public override string ToString() => IsNothing ? default(T) == null ? "Nothing" :
            ↪ default(T).ToString() : IsExternal ? $"<{AbsoluteValue}>" : Value.ToString();
76      }
77  }
```

./ILinks.cs

```
1   using Platform.Data.Constants;
2
3   namespace Platform.Data.Doublets
4   {
5       public interface ILinks<TLink> : ILinks<TLink, LinksCombinedConstants<TLink, TLink, int>>
6       {
7       }
8   }
```

./ILinksExtensions.cs

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

```
121             {
122                 var next = path[i];
123                 var values = links.GetLink(current);
124                 var source = values[constants.SourcePart];
125                 var target = values[constants.TargetPart];
126                 if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
                    ↪ next))
127                 {
128                     //throw new Exception(string.Format("Невозможно выбрать путь, так как и
                        ↪ Source и Target совпадают с элементом пути {0}.", next));
129                     return false;
130                 }
131                 if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
                    ↪ target))
132                 {
133                     //throw new Exception(string.Format("Невозможно продолжить путь через
                        ↪ элемент пути {0}", next));
134                     return false;
135                 }
136                 current = next;
137             }
138             return true;
139         }
140 
141         /// <remarks>
142         /// Может потребовать дополнительного стека для PathElement's при использовании
            ↪ SequenceWalker.
143         /// </remarks>
144         public static TLink GetByKeys<TLink>(this ILinks<TLink> links, TLink root, params int[]
            ↪ path)
145         {
146             links.EnsureLinkExists(root, "root");
147             var currentLink = root;
148             for (var i = 0; i < path.Length; i++)
149             {
150                 currentLink = links.GetLink(currentLink)[path[i]];
151             }
152             return currentLink;
153         }
154 
155         public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this ILinks<TLink>
            ↪ links, TLink root, ulong size, ulong index)
156         {
157             var constants = links.Constants;
158             var source = constants.SourcePart;
159             var target = constants.TargetPart;
160             if (!MathHelpers.IsPowerOfTwo(size))
161             {
162                 throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes other
                    ↪ than powers of two are not supported.");
163             }
164             var path = new BitArray(BitConverter.GetBytes(index));
165             var length = BitwiseHelpers.GetLowestBitPosition(size);
166             links.EnsureLinkExists(root, "root");
167             var currentLink = root;
168             for (var i = length - 1; i >= 0; i--)
169             {
170                 currentLink = links.GetLink(currentLink)[path[i] ? target : source];
171             }
172             return currentLink;
173         }
174 
175         #endregion
176 
177         /// <summary>
178         /// Возвращает индекс указанной связи.
179         /// </summary>
180         /// <param name="links">Хранилище связей.</param>
181         /// <param name="link">Связь представленная списком, состоящим из её адреса и
            ↪ содержимого.</param>
182         /// <returns>Индекс начальной связи для указанной связи.</returns>
183         [MethodImpl(MethodImplOptions.AggressiveInlining)]
184         public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
            ↪ link[links.Constants.IndexPart];
185 
186         /// <summary>
187         /// Возвращает индекс начальной (Source) связи для указанной связи.
188         /// </summary>
```

```
189         /// <param name="links">Хранилище связей.</param>
190         /// <param name="link">Индекс связи.</param>
191         /// <returns>Индекс начальной связи для указанной связи.</returns>
192         [MethodImpl(MethodImplOptions.AggressiveInlining)]
193         public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink link) =>
            ↪ links.GetLink(link)[links.Constants.SourcePart];
194
195         /// <summary>
196         /// Возвращает индекс начальной (Source) связи для указанной связи.
197         /// </summary>
198         /// <param name="links">Хранилище связей.</param>
199         /// <param name="link">Связь представленная списком, состоящим из её адреса и
            ↪ содержимого.</param>
200         /// <returns>Индекс начальной связи для указанной связи.</returns>
201         [MethodImpl(MethodImplOptions.AggressiveInlining)]
202         public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
            ↪ link[links.Constants.SourcePart];
203
204         /// <summary>
205         /// Возвращает индекс конечной (Target) связи для указанной связи.
206         /// </summary>
207         /// <param name="links">Хранилище связей.</param>
208         /// <param name="link">Индекс связи.</param>
209         /// <returns>Индекс конечной связи для указанной связи.</returns>
210         [MethodImpl(MethodImplOptions.AggressiveInlining)]
211         public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
            ↪ links.GetLink(link)[links.Constants.TargetPart];
212
213         /// <summary>
214         /// Возвращает индекс конечной (Target) связи для указанной связи.
215         /// </summary>
216         /// <param name="links">Хранилище связей.</param>
217         /// <param name="link">Связь представленная списком, состоящим из её адреса и
            ↪ содержимого.</param>
218         /// <returns>Индекс конечной связи для указанной связи.</returns>
219         [MethodImpl(MethodImplOptions.AggressiveInlining)]
220         public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
            ↪ link[links.Constants.TargetPart];
221
222         /// <summary>
223         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪ (handler) для каждой подходящей связи.
224         /// </summary>
225         /// <param name="links">Хранилище связей.</param>
226         /// <param name="handler">Обработчик каждой подходящей связи.</param>
227         /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
            ↪ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
            ↪ Any - отсутствие ограничения, 1..∞ конкретный адрес связи.</param>
228         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪ случае.</returns>
229         [MethodImpl(MethodImplOptions.AggressiveInlining)]
230         public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
            ↪ handler, params TLink[] restrictions)
231             => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
                ↪ links.Constants.Continue);
232
233         /// <summary>
234         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪ (handler) для каждой подходящей связи.
235         /// </summary>
236         /// <param name="links">Хранилище связей.</param>
237         /// <param name="source">Значение, определяющее соответствующие шаблону связи.
            ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
            ↪ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
238         /// <param name="target">Значение, определяющее соответствующие шаблону связи.
            ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
            ↪ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
239         /// <param name="handler">Обработчик каждой подходящей связи.</param>
240         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪ случае.</returns>
241         [MethodImpl(MethodImplOptions.AggressiveInlining)]
242         public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
            ↪ Func<TLink, bool> handler)
243         {
244             var constants = links.Constants;
245             return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
                ↪ constants.Break, constants.Any, source, target);
```

```
246         }
247
248         /// <summary>
249         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪ (handler) для каждой подходящей связи.
250         /// </summary>
251         /// <param name="links">Хранилище связей.</param>
252         /// <param name="source">Значение, определяющее соответствующие шаблону связи.
            ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
            ↪ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
253         /// <param name="target">Значение, определяющее соответствующие шаблону связи.
            ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
            ↪ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
254         /// <param name="handler">Обработчик каждой подходящей связи.</param>
255         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪ случае.</returns>
256         [MethodImpl(MethodImplOptions.AggressiveInlining)]
257         public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
            ↪ Func<IList<TLink>, TLink> handler)
258         {
259             var constants = links.Constants;
260             return links.Each(handler, constants.Any, source, target);
261         }
262
263         [MethodImpl(MethodImplOptions.AggressiveInlining)]
264         public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
            ↪ restrictions)
265         {
266             var constants = links.Constants;
267             int listSize = (Integer<TLink>)links.Count(restrictions);
268             var list = new IList<TLink>[listSize];
269             if (listSize > 0)
270             {
271                 var filler = new ArrayFiller<IList<TLink>, TLink>(list,
                    ↪ links.Constants.Continue);
272                 links.Each(filler.AddAndReturnConstant, restrictions);
273             }
274             return list;
275         }
276
277         /// <summary>
278         /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
            ↪ в хранилище связей.
279         /// </summary>
280         /// <param name="links">Хранилище связей.</param>
281         /// <param name="source">Начало связи.</param>
282         /// <param name="target">Конец связи.</param>
283         /// <returns>Значение, определяющее существует ли связь.</returns>
284         [MethodImpl(MethodImplOptions.AggressiveInlining)]
285         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;
286
287         #region Ensure
288         // TODO: May be move to EnsureExtensions or make it both there and here
289
290         [MethodImpl(MethodImplOptions.AggressiveInlining)]
291         public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
            ↪ reference, string argumentName)
292         {
293             if (links.IsInnerReference(reference) && !links.Exists(reference))
294             {
295                 throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
296             }
297         }
298
299         [MethodImpl(MethodImplOptions.AggressiveInlining)]
300         public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
            ↪ IList<TLink> restrictions, string argumentName)
301         {
302             for (int i = 0; i < restrictions.Count; i++)
303             {
304                 links.EnsureInnerReferenceExists(restrictions[i], argumentName);
305             }
306         }
307
308         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
309         public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
            ↪ restrictions)
310         {
311             for (int i = 0; i < restrictions.Count; i++)
312             {
313                 links.EnsureLinkIsAnyOrExists(restrictions[i], nameof(restrictions));
314             }
315         }
316 
317         [MethodImpl(MethodImplOptions.AggressiveInlining)]
318         public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
            ↪ string argumentName)
319         {
320             var equalityComparer = EqualityComparer<TLink>.Default;
321             if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
322             {
323                 throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
324             }
325         }
326 
327         [MethodImpl(MethodImplOptions.AggressiveInlining)]
328         public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links, TLink
            ↪ link, string argumentName)
329         {
330             var equalityComparer = EqualityComparer<TLink>.Default;
331             if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
332             {
333                 throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
334             }
335         }
336 
337         /// <param name="links">Хранилище связей.</param>
338         [MethodImpl(MethodImplOptions.AggressiveInlining)]
339         public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink source,
            ↪ TLink target)
340         {
341             if (links.Exists(source, target))
342             {
343                 throw new LinkWithSameValueAlreadyExistsException();
344             }
345         }
346 
347         /// <param name="links">Хранилище связей.</param>
348         public static void EnsureNoDependencies<TLink>(this ILinks<TLink> links, TLink link)
349         {
350             if (links.DependenciesExist(link))
351             {
352                 throw new ArgumentLinkHasDependenciesException<TLink>(link);
353             }
354         }
355 
356         /// <param name="links">Хранилище связей.</param>
357         public static void EnsureCreated<TLink>(this ILinks<TLink> links, params TLink[]
            ↪ addresses) => links.EnsureCreated(links.Create, addresses);
358 
359         /// <param name="links">Хранилище связей.</param>
360         public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params TLink[]
            ↪ addresses) => links.EnsureCreated(links.CreatePoint, addresses);
361 
362         /// <param name="links">Хранилище связей.</param>
363         public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
            ↪ params TLink[] addresses)
364         {
365             var constants = links.Constants;
366             var nonExistentAddresses = new HashSet<ulong>(addresses.Where(x =>
                ↪ !links.Exists(x)).Select(x => (ulong)(Integer<TLink>)x));
367             if (nonExistentAddresses.Count > 0)
368             {
369                 var max = nonExistentAddresses.Max();
370                 // TODO: Эту верхнюю границу нужно разрешить переопределять (проверить
                    ↪ применяется ли эта логика)
371                 max = Math.Min(max, (Integer<TLink>)constants.MaxPossibleIndex);
372                 var createdLinks = new List<TLink>();
373                 var equalityComparer = EqualityComparer<TLink>.Default;
374                 TLink createdLink = creator();
375                 while (!equalityComparer.Equals(createdLink, (Integer<TLink>)max))
376                 {
377                     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);
    }

```

```
448         /// <param name="links">Хранилище связей.</param>
449         [MethodImpl(MethodImplOptions.AggressiveInlining)]
450         public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
            ↪ target) => links.Update(links.Create(), source, target);
451
452         /// <summary>
453         /// Обновляет связь с указанными началом (Source) и концом (Target)
454         /// на связь с указанными началом (NewSource) и концом (NewTarget).
455         /// </summary>
456         /// <param name="links">Хранилище связей.</param>
457         /// <param name="link">Индекс обновляемой связи.</param>
458         /// <param name="newSource">Индекс связи, которая является началом связи, на которую
            ↪ выполняется обновление.</param>
459         /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
            ↪ выполняется обновление.</param>
460         /// <returns>Индекс обновлённой связи.</returns>
461         [MethodImpl(MethodImplOptions.AggressiveInlining)]
462         public static TLink Update<TLink>(this ILinks<TLink> links, TLink link, TLink newSource,
            ↪ TLink newTarget) => links.Update(new[] { link, newSource, newTarget });
463
464         /// <summary>
465         /// Обновляет связь с указанными началом (Source) и концом (Target)
466         /// на связь с указанными началом (NewSource) и концом (NewTarget).
467         /// </summary>
468         /// <param name="links">Хранилище связей.</param>
469         /// <param name="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>
470         /// <returns>Индекс обновлённой связи.</returns>
471         [MethodImpl(MethodImplOptions.AggressiveInlining)]
472         public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
473         {
474             if (restrictions.Length == 2)
475             {
476                 return links.Merge(restrictions[0], restrictions[1]);
477             }
478             if (restrictions.Length == 4)
479             {
480                 return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
                    ↪ restrictions[2], restrictions[3]);
481             }
482             else
483             {
484                 return links.Update(restrictions);
485             }
486         }
487
488         /// <summary>
489         /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
            ↪ с указанными Source (началом) и Target (концом).
490         /// </summary>
491         /// <param name="links">Хранилище связей.</param>
492         /// <param name="source">Индекс связи, которая является началом на создаваемой
            ↪ связи.</param>
493         /// <param name="target">Индекс связи, которая является концом для создаваемой
            ↪ связи.</param>
494         /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
495         [MethodImpl(MethodImplOptions.AggressiveInlining)]
496         public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
            ↪ target)
497         {
498             var link = links.SearchOrDefault(source, target);
499             if (EqualityComparer<TLink>.Default.Equals(link, default))
500             {
501                 link = links.CreateAndUpdate(source, target);
502             }
503             return link;
504         }
505
506         /// <summary>
507         /// Обновляет связь с указанными началом (Source) и концом (Target)
508         /// на связь с указанными началом (NewSource) и концом (NewTarget).
509         /// </summary>
510         /// <param name="links">Хранилище связей.</param>
511         /// <param name="source">Индекс связи, которая является началом обновляемой
            ↪ связи.</param>
```

```
512         /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
513         /// <param name="newSource">Индекс связи, которая является началом связи, на которую
            ↪ выполняется обновление.</param>
514         /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
            ↪ выполняется обновление.</param>
515         /// <returns>Индекс обновлённой связи.</returns>
516         [MethodImpl(MethodImplOptions.AggressiveInlining)]
517         public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
            ↪ TLink target, TLink newSource, TLink newTarget)
518         {
519             var equalityComparer = EqualityComparer<TLink>.Default;
520             var link = links.SearchOrDefault(source, target);
521             if (equalityComparer.Equals(link, default))
522             {
523                 return links.CreateAndUpdate(newSource, newTarget);
524             }
525             if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
                ↪ target))
526             {
527                 return link;
528             }
529             return links.Update(link, newSource, newTarget);
530         }
531 
532         /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
533         /// <param name="links">Хранилище связей.</param>
534         /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
535         /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
536         [MethodImpl(MethodImplOptions.AggressiveInlining)]
537         public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
            ↪ target)
538         {
539             var link = links.SearchOrDefault(source, target);
540             if (!EqualityComparer<TLink>.Default.Equals(link, default))
541             {
542                 links.Delete(link);
543                 return link;
544             }
545             return default;
546         }
547 
548         /// <summary>Удаляет несколько связей.</summary>
549         /// <param name="links">Хранилище связей.</param>
550         /// <param name="deletedLinks">Список адресов связей к удалению.</param>
551         [MethodImpl(MethodImplOptions.AggressiveInlining)]
552         public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
553         {
554             for (int i = 0; i < deletedLinks.Count; i++)
555             {
556                 links.Delete(deletedLinks[i]);
557             }
558         }
559 
560         // Replace one link with another (replaced link is deleted, children are updated or
            ↪ deleted)
561         public static TLink Merge<TLink>(this ILinks<TLink> links, TLink linkIndex, TLink
            ↪ newLink)
562         {
563             var equalityComparer = EqualityComparer<TLink>.Default;
564             if (equalityComparer.Equals(linkIndex, newLink))
565             {
566                 return newLink;
567             }
568             var constants = links.Constants;
569             ulong referencesAsSourceCount = (Integer<TLink>)links.Count(constants.Any,
                ↪ linkIndex, constants.Any);
570             ulong referencesAsTargetCount = (Integer<TLink>)links.Count(constants.Any,
                ↪ constants.Any, linkIndex);
571             var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(linkIndex)) &&
                ↪ referencesAsSourceCount == 1 && referencesAsTargetCount == 1;
572             if (!isStandalonePoint)
573             {
574                 var totalReferences = referencesAsSourceCount + referencesAsTargetCount;
575                 if (totalReferences > 0)
576                 {
577                     var references = ArrayPool.Allocate<TLink>((long)totalReferences);
578                     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

```
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

```
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

```
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

```
using Platform.Data.Constants;

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

./Link.cs

```
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);

```

```
69          public static string ToString(TLink index, TLink source, TLink target) => $"({index}:
            ↪ {source}->{target})";
70
71          public static string ToString(TLink source, TLink target) => $"({source}->{target})";
72
73          public static implicit operator TLink[](Link<TLink> link) => link.ToArray();
74
75          public static implicit operator Link<TLink>(TLink[] linkArray) => new
            ↪ Link<TLink>(linkArray);
76
77          public TLink[] ToArray()
78          {
79              var array = new TLink[Length];
80              CopyTo(array, 0);
81              return array;
82          }
83
84          public override string ToString() => _equalityComparer.Equals(Index, _constants.Null) ?
            ↪ ToString(Source, Target) : ToString(Index, Source, Target);
85
86          #region IList
87
88          public int Count => Length;
89
90          public bool IsReadOnly => true;
91
92          public TLink this[int index]
93          {
94              get
95              {
96                  Ensure.Always.ArgumentInRange(index, new Range<int>(0, Length - 1),
                    ↪ nameof(index));
97                  if (index == _constants.IndexPart)
98                  {
99                      return Index;
100                 }
101                 if (index == _constants.SourcePart)
102                 {
103                     return Source;
104                 }
105                 if (index == _constants.TargetPart)
106                 {
107                     return Target;
108                 }
109                 throw new NotSupportedException(); // Impossible path due to
                    ↪ Ensure.ArgumentInRange
110             }
111             set => throw new NotSupportedException();
112         }
113
114         IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
115
116         public IEnumerator<TLink> GetEnumerator()
117         {
118             yield return Index;
119             yield return Source;
120             yield return Target;
121         }
122
123         public void Add(TLink item) => throw new NotSupportedException();
124
125         public void Clear() => throw new NotSupportedException();
126
127         public bool Contains(TLink item) => IndexOf(item) >= 0;
128
129         public void CopyTo(TLink[] array, int arrayIndex)
130         {
131             Ensure.Always.ArgumentNotNull(array, nameof(array));
132             Ensure.Always.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
                ↪ nameof(arrayIndex));
133             if (arrayIndex + Length > array.Length)
134             {
135                 throw new InvalidOperationException();
136             }
137             array[arrayIndex++] = Index;
138             array[arrayIndex++] = Source;
139             array[arrayIndex] = Target;
140         }
141
```

```
142         public bool Remove(TLink item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
143
144         public int IndexOf(TLink item)
145         {
146             if (_equalityComparer.Equals(Index, item))
147             {
148                 return _constants.IndexPart;
149             }
150             if (_equalityComparer.Equals(Source, item))
151             {
152                 return _constants.SourcePart;
153             }
154             if (_equalityComparer.Equals(Target, item))
155             {
156                 return _constants.TargetPart;
157             }
158             return -1;
159         }
160
161         public void Insert(int index, TLink item) => throw new NotSupportedException();
162
163         public void RemoveAt(int index) => throw new NotSupportedException();
164
165         #endregion
166     }
167 }
```

./LinkExtensions.cs

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

./LinksOperatorBase.cs

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

./PropertyOperators/DefaultLinkPropertyOperator.cs

```
1  using System.Linq;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4
5  namespace Platform.Data.Doublets.PropertyOperators
6  {
7      public class DefaultLinkPropertyOperator<TLink> : LinksOperatorBase<TLink>,
       ↪  IPropertyOperator<TLink, TLink, TLink>
8      {
9          private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
10
11         public DefaultLinkPropertyOperator(ILinks<TLink> links) : base(links)
12         {
13         }
14
15         public TLink GetValue(TLink @object, TLink property)
16         {
17             var objectProperty = Links.SearchOrDefault(@object, property);
18             if (_equalityComparer.Equals(objectProperty, default))
19             {
20                 return default;
21             }
22             var valueLink = Links.All(Links.Constants.Any, objectProperty).SingleOrDefault();
23             if (valueLink == null)
24             {
25                 return default;
26             }
27             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

```
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

```
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)]
```

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

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

```
176         /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
            ↪ байтах.</param>
177         public ResizableDirectMemoryLinks(string address, long memoryReservationStep)
178             : this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
                ↪ memoryReservationStep)
179         {
180         }
181
182         public ResizableDirectMemoryLinks(IResizableDirectMemory memory)
183             : this(memory, DefaultLinksSizeStep)
184         {
185         }
186
187         public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
            ↪ memoryReservationStep)
188         {
189             Constants = Default<LinksCombinedConstants<TLink, TLink, int>>.Instance;
190             _memory = memory;
191             _memoryReservationStep = memoryReservationStep;
192             if (memory.ReservedCapacity < memoryReservationStep)
193             {
194                 memory.ReservedCapacity = memoryReservationStep;
195             }
196             SetPointers(_memory);
197             // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
198             _memory.UsedCapacity = ((long)(Integer<TLink>)LinksHeader.GetAllocatedLinks(_header)
                ↪ * LinkSizeInBytes) + LinkHeaderSizeInBytes;
199             // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
200             LinksHeader.SetReservedLinks(_header, (Integer<TLink>)((_memory.ReservedCapacity -
                ↪ LinkHeaderSizeInBytes) / LinkSizeInBytes));
201         }
202
203         [MethodImpl(MethodImplOptions.AggressiveInlining)]
204         public TLink Count(IList<TLink> restrictions)
205         {
206             // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
207             if (restrictions.Count == 0)
208             {
209                 return Total;
210             }
211             if (restrictions.Count == 1)
212             {
213                 var index = restrictions[Constants.IndexPart];
214                 if (_equalityComparer.Equals(index, Constants.Any))
215                 {
216                     return Total;
217                 }
218                 return Exists(index) ? Integer<TLink>.One : Integer<TLink>.Zero;
219             }
220             if (restrictions.Count == 2)
221             {
222                 var index = restrictions[Constants.IndexPart];
223                 var value = restrictions[1];
224                 if (_equalityComparer.Equals(index, Constants.Any))
225                 {
226                     if (_equalityComparer.Equals(value, Constants.Any))
227                     {
228                         return Total; // Any - как отсутствие ограничения
229                     }
230                     return Add(_sourcesTreeMethods.CalculateReferences(value),
                        ↪ _targetsTreeMethods.CalculateReferences(value));
231                 }
232                 else
233                 {
234                     if (!Exists(index))
235                     {
236                         return Integer<TLink>.Zero;
237                     }
238                     if (_equalityComparer.Equals(value, Constants.Any))
239                     {
240                         return Integer<TLink>.One;
241                     }
242                     var storedLinkValue = GetLinkUnsafe(index);
243                     if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
244                         _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
245                     {
246                         return Integer<TLink>.One;
247                     }
```

```
248                     return Integer<TLink>.Zero;
249                 }
250             }
251             if (restrictions.Count == 3)
252             {
253                 var index = restrictions[Constants.IndexPart];
254                 var source = restrictions[Constants.SourcePart];
255                 var target = restrictions[Constants.TargetPart];
256 
257                 if (_equalityComparer.Equals(index, Constants.Any))
258                 {
259                     if (_equalityComparer.Equals(source, Constants.Any) &&
                        ↪ _equalityComparer.Equals(target, Constants.Any))
260                     {
261                         return Total;
262                     }
263                     else if (_equalityComparer.Equals(source, Constants.Any))
264                     {
265                         return _targetsTreeMethods.CalculateReferences(target);
266                     }
267                     else if (_equalityComparer.Equals(target, Constants.Any))
268                     {
269                         return _sourcesTreeMethods.CalculateReferences(source);
270                     }
271                     else //if(source != Any && target != Any)
272                     {
273                         // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
274                         var link = _sourcesTreeMethods.Search(source, target);
275                         return _equalityComparer.Equals(link, Constants.Null) ?
                            ↪ Integer<TLink>.Zero : Integer<TLink>.One;
276                     }
277                 }
278                 else
279                 {
280                     if (!Exists(index))
281                     {
282                         return Integer<TLink>.Zero;
283                     }
284                     if (_equalityComparer.Equals(source, Constants.Any) &&
                        ↪ _equalityComparer.Equals(target, Constants.Any))
285                     {
286                         return Integer<TLink>.One;
287                     }
288                     var storedLinkValue = GetLinkUnsafe(index);
289                     if (!_equalityComparer.Equals(source, Constants.Any) &&
                        ↪ !_equalityComparer.Equals(target, Constants.Any))
290                     {
291                         if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source) &&
292                             _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
293                         {
294                             return Integer<TLink>.One;
295                         }
296                         return Integer<TLink>.Zero;
297                     }
298                     var value = default(TLink);
299                     if (_equalityComparer.Equals(source, Constants.Any))
300                     {
301                         value = target;
302                     }
303                     if (_equalityComparer.Equals(target, Constants.Any))
304                     {
305                         value = source;
306                     }
307                     if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
308                         _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
309                     {
310                         return Integer<TLink>.One;
311                     }
312                     return Integer<TLink>.Zero;
313                 }
314             }
315             throw new NotSupportedException("Другие размеры и способы ограничений не
                ↪ поддерживаются.");
316         }
317 
318         [MethodImpl(MethodImplOptions.AggressiveInlining)]
319         public TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
320         {
```

```
321             if (restrictions.Count == 0)
322             {
323                 for (TLink link = Integer<TLink>.One; _comparer.Compare(link,
                    ↪ (Integer<TLink>)LinksHeader.GetAllocatedLinks(_header)) <= 0; link =
                    ↪ Increment(link))
324                 {
325                     if (Exists(link) && _equalityComparer.Equals(handler(GetLinkStruct(link)),
                        ↪ Constants.Break))
326                     {
327                         return Constants.Break;
328                     }
329                 }
330 
331                 return Constants.Continue;
332             }
333             if (restrictions.Count == 1)
334             {
335                 var index = restrictions[Constants.IndexPart];
336                 if (_equalityComparer.Equals(index, Constants.Any))
337                 {
338                     return Each(handler, ArrayPool<TLink>.Empty);
339                 }
340                 if (!Exists(index))
341                 {
342                     return Constants.Continue;
343                 }
344                 return handler(GetLinkStruct(index));
345             }
346             if (restrictions.Count == 2)
347             {
348                 var index = restrictions[Constants.IndexPart];
349                 var value = restrictions[1];
350                 if (_equalityComparer.Equals(index, Constants.Any))
351                 {
352                     if (_equalityComparer.Equals(value, Constants.Any))
353                     {
354                         return Each(handler, ArrayPool<TLink>.Empty);
355                     }
356                     if (_equalityComparer.Equals(Each(handler, new[] { index, value,
                        ↪ Constants.Any }), Constants.Break))
357                     {
358                         return Constants.Break;
359                     }
360                     return Each(handler, new[] { index, Constants.Any, value });
361                 }
362                 else
363                 {
364                     if (!Exists(index))
365                     {
366                         return Constants.Continue;
367                     }
368                     if (_equalityComparer.Equals(value, Constants.Any))
369                     {
370                         return handler(GetLinkStruct(index));
371                     }
372                     var storedLinkValue = GetLinkUnsafe(index);
373                     if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
374                         _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
375                     {
376                         return handler(GetLinkStruct(index));
377                     }
378                     return Constants.Continue;
379                 }
380             }
381             if (restrictions.Count == 3)
382             {
383                 var index = restrictions[Constants.IndexPart];
384                 var source = restrictions[Constants.SourcePart];
385                 var target = restrictions[Constants.TargetPart];
386                 if (_equalityComparer.Equals(index, Constants.Any))
387                 {
388                     if (_equalityComparer.Equals(source, Constants.Any) &&
                        ↪ _equalityComparer.Equals(target, Constants.Any))
389                     {
390                         return Each(handler, ArrayPool<TLink>.Empty);
391                     }
392                     else if (_equalityComparer.Equals(source, Constants.Any))
393                     {
```

```
394                         return _targetsTreeMethods.EachReference(target, handler);
395                     }
396                     else if (_equalityComparer.Equals(target, Constants.Any))
397                     {
398                         return _sourcesTreeMethods.EachReference(source, handler);
399                     }
400                     else //if(source != Any && target != Any)
401                     {
402                         var link = _sourcesTreeMethods.Search(source, target);
403                         return _equalityComparer.Equals(link, Constants.Null) ?
                            ↪ Constants.Continue : handler(GetLinkStruct(link));
404                     }
405                 }
406                 else
407                 {
408                     if (!Exists(index))
409                     {
410                         return Constants.Continue;
411                     }
412                     if (_equalityComparer.Equals(source, Constants.Any) &&
                        ↪ _equalityComparer.Equals(target, Constants.Any))
413                     {
414                         return handler(GetLinkStruct(index));
415                     }
416                     var storedLinkValue = GetLinkUnsafe(index);
417                     if (!_equalityComparer.Equals(source, Constants.Any) &&
                        ↪ !_equalityComparer.Equals(target, Constants.Any))
418                     {
419                         if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source) &&
420                             _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
421                         {
422                             return handler(GetLinkStruct(index));
423                         }
424                         return Constants.Continue;
425                     }
426                     var value = default(TLink);
427                     if (_equalityComparer.Equals(source, Constants.Any))
428                     {
429                         value = target;
430                     }
431                     if (_equalityComparer.Equals(target, Constants.Any))
432                     {
433                         value = source;
434                     }
435                     if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
436                         _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
437                     {
438                         return handler(GetLinkStruct(index));
439                     }
440                     return Constants.Continue;
441                 }
442             }
443             throw new NotSupportedException("Другие размеры и способы ограничений не
                ↪ поддерживаются.");
444         }
445 
446         /// <remarks>
447         /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
                ↪ в другом месте (но не в менеджере памяти, а в логике Links)
448         /// </remarks>
449         [MethodImpl(MethodImplOptions.AggressiveInlining)]
450         public TLink Update(IList<TLink> values)
451         {
452             var linkIndex = values[Constants.IndexPart];
453             var link = GetLinkUnsafe(linkIndex);
454             // Будет корректно работать только в том случае, если пространство выделенной связи
                ↪ предварительно заполнено нулями
455             if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
456             {
457                 _sourcesTreeMethods.Detach(LinksHeader.GetFirstAsSourcePointer(_header),
                    ↪ linkIndex);
458             }
459             if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
460             {
461                 _targetsTreeMethods.Detach(LinksHeader.GetFirstAsTargetPointer(_header),
                    ↪ linkIndex);
462             }
463             Link.SetSource(link, values[Constants.SourcePart]);
```

```
464             Link.SetTarget(link, values[Constants.TargetPart]);
465             if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
466             {
467                 _sourcesTreeMethods.Attach(LinksHeader.GetFirstAsSourcePointer(_header),
                    ↪ linkIndex);
468             }
469             if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
470             {
471                 _targetsTreeMethods.Attach(LinksHeader.GetFirstAsTargetPointer(_header),
                    ↪ linkIndex);
472             }
473             return linkIndex;
474         }
475
476         [MethodImpl(MethodImplOptions.AggressiveInlining)]
477         public Link<TLink> GetLinkStruct(TLink linkIndex)
478         {
479             var link = GetLinkUnsafe(linkIndex);
480             return new Link<TLink>(linkIndex, Link.GetSource(link), Link.GetTarget(link));
481         }
482
483         [MethodImpl(MethodImplOptions.AggressiveInlining)]
484         private IntPtr GetLinkUnsafe(TLink linkIndex) => _links.GetElement(LinkSizeInBytes,
            ↪ linkIndex);
485
486         /// <remarks>
487         /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
            ↪ пространство
488         /// </remarks>
489         public TLink Create()
490         {
491             var freeLink = LinksHeader.GetFirstFreeLink(_header);
492             if (!_equalityComparer.Equals(freeLink, Constants.Null))
493             {
494                 _unusedLinksListMethods.Detach(freeLink);
495             }
496             else
497             {
498                 if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
                    ↪ Constants.MaxPossibleIndex) > 0)
499                 {
500                     throw new
                        ↪ LinksLimitReachedException((Integer<TLink>)Constants.MaxPossibleIndex);
501                 }
502                 if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
                    ↪ Decrement(LinksHeader.GetReservedLinks(_header))) >= 0)
503                 {
504                     _memory.ReservedCapacity += _memoryReservationStep;
505                     SetPointers(_memory);
506                     LinksHeader.SetReservedLinks(_header,
                        ↪ (Integer<TLink>)(_memory.ReservedCapacity / LinkSizeInBytes));
507                 }
508                 LinksHeader.SetAllocatedLinks(_header,
                    ↪ Increment(LinksHeader.GetAllocatedLinks(_header)));
509                 _memory.UsedCapacity += LinkSizeInBytes;
510                 freeLink = LinksHeader.GetAllocatedLinks(_header);
511             }
512             return freeLink;
513         }
514
515         public void Delete(TLink link)
516         {
517             if (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) < 0)
518             {
519                 _unusedLinksListMethods.AttachAsFirst(link);
520             }
521             else if (_equalityComparer.Equals(link, LinksHeader.GetAllocatedLinks(_header)))
522             {
523                 LinksHeader.SetAllocatedLinks(_header,
                    ↪ Decrement(LinksHeader.GetAllocatedLinks(_header)));
524                 _memory.UsedCapacity -= LinkSizeInBytes;
525                 // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
                    ↪ пока не дойдём до первой существующей связи
526                 // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
527                 while ((_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
                    ↪ Integer<TLink>.Zero) > 0) &&
                    ↪ IsUnusedLink(LinksHeader.GetAllocatedLinks(_header)))
528                 {
```

```
                    _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

```
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

```
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);
```

```
179
180             protected override void SetSize(TLink node, TLink size)
181             {
182                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsSourceOffset).GetValue<TLink>();
183                 (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsSourceOffset).SetValue(BitwiseHelpers.PartialWrite(previousValue,
                    ↪  size, 5, -5));
184             }
185
186             protected override bool GetLeftIsChild(TLink node)
187             {
188                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsSourceOffset).GetValue<TLink>();
189                 return (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 4, 1);
190             }
191
192             protected override void SetLeftIsChild(TLink node, bool value)
193             {
194                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsSourceOffset).GetValue<TLink>();
195                 var modified = BitwiseHelpers.PartialWrite(previousValue,
                    ↪  (TLink)(Integer<TLink>)value, 4, 1);
196                 (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsSourceOffset).SetValue(modified);
197             }
198
199             protected override bool GetRightIsChild(TLink node)
200             {
201                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsSourceOffset).GetValue<TLink>();
202                 return (Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 3, 1);
203             }
204
205             protected override void SetRightIsChild(TLink node, bool value)
206             {
207                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsSourceOffset).GetValue<TLink>();
208                 var modified = BitwiseHelpers.PartialWrite(previousValue,
                    ↪  (TLink)(Integer<TLink>)value, 3, 1);
209                 (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsSourceOffset).SetValue(modified);
210             }
211
212             protected override sbyte GetBalance(TLink node)
213             {
214                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsSourceOffset).GetValue<TLink>();
215                 var value = (ulong)(Integer<TLink>)BitwiseHelpers.PartialRead(previousValue, 0,
                    ↪  3);
216                 var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
                    ↪  124 : value & 3);
217                 return unpackedValue;
218             }
219
220             protected override void SetBalance(TLink node, sbyte value)
221             {
222                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsSourceOffset).GetValue<TLink>();
223                 var packagedValue = (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) | value &
                    ↪  3);
224                 var modified = BitwiseHelpers.PartialWrite(previousValue, packagedValue, 0, 3);
225                 (Links.GetElement(LinkSizeInBytes, node) +
                    ↪  Link.SizeAsSourceOffset).SetValue(modified);
226             }
227
228             protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
229             {
230                 var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
                    ↪  Link.SourceOffset).GetValue<TLink>();
231                 var secondSource = (Links.GetElement(LinkSizeInBytes, second) +
                    ↪  Link.SourceOffset).GetValue<TLink>();
232                 return LessThan(firstSource, secondSource) ||
```

```
233                     (IsEquals(firstSource, secondSource) &&
                            LessThan((Links.GetElement(LinkSizeInBytes, first) +
                        ↪   Link.TargetOffset).GetValue<TLink>(),
                        ↪   (Links.GetElement(LinkSizeInBytes, second) +
                        ↪   Link.TargetOffset).GetValue<TLink>()));
234             }
235
236             protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
237             {
238                 var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
                    ↪   Link.SourceOffset).GetValue<TLink>();
239                 var secondSource = (Links.GetElement(LinkSizeInBytes, second) +
                    ↪   Link.SourceOffset).GetValue<TLink>();
240                 return GreaterThan(firstSource, secondSource) ||
241                        (IsEquals(firstSource, secondSource) &&
                            GreaterThan((Links.GetElement(LinkSizeInBytes, first) +
                        ↪   Link.TargetOffset).GetValue<TLink>(),
                        ↪   (Links.GetElement(LinkSizeInBytes, second) +
                        ↪   Link.TargetOffset).GetValue<TLink>()));
242             }
243
244             protected override TLink GetTreeRoot() => (Header +
                ↪   LinksHeader.FirstAsSourceOffset).GetValue<TLink>();
245
246             protected override TLink GetBasePartValue(TLink link) =>
                ↪   (Links.GetElement(LinkSizeInBytes, link) + Link.SourceOffset).GetValue<TLink>();
247
248             /// <summary>
249             /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
                ↪   (концом)
250             /// по дереву (индексу) связей, отсортированному по Source, а затем по Target.
251             /// </summary>
252             /// <param name="source">Индекс связи, которая является началом на искомой
                ↪   связи.</param>
253             /// <param name="target">Индекс связи, которая является концом на искомой
                ↪   связи.</param>
254             /// <returns>Индекс искомой связи.</returns>
255             public TLink Search(TLink source, TLink target)
256             {
257                 var root = GetTreeRoot();
258                 while (!EqualToZero(root))
259                 {
260                     var rootSource = (Links.GetElement(LinkSizeInBytes, root) +
                        ↪   Link.SourceOffset).GetValue<TLink>();
261                     var rootTarget = (Links.GetElement(LinkSizeInBytes, root) +
                        ↪   Link.TargetOffset).GetValue<TLink>();
262                     if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                        ↪   node.Key < root.Key
263                     {
264                         root = GetLeftOrDefault(root);
265                     }
266                     else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget))
                        ↪   // node.Key > root.Key
267                     {
268                         root = GetRightOrDefault(root);
269                     }
270                     else // node.Key == root.Key
271                     {
272                         return root;
273                     }
274                 }
275                 return GetZero();
276             }
277
278             [MethodImpl(MethodImplOptions.AggressiveInlining)]
279             private bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget, TLink
                ↪   secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
                ↪   (IsEquals(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
280
281             [MethodImpl(MethodImplOptions.AggressiveInlining)]
282             private bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget, TLink
                ↪   secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
                ↪   (IsEquals(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
283         }
284
285         private class LinksTargetsTreeMethods : LinksTreeMethodsBase
286         {
```

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

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

./ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Disposables;
5  using Platform.Collections.Arrays;
6  using Platform.Helpers.Singletons;
7  using Platform.Memory;
8  using Platform.Data.Exceptions;
9  using Platform.Data.Constants;
10 
11 //#define ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
12 
13 #pragma warning disable 0649
14 #pragma warning disable 169
15 
16 // ReSharper disable BuiltInTypeReferenceStyle
17 
18 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;
```

```
91              _memory = memory;
92              _memoryReservationStep = memoryReservationStep;
93              if (memory.ReservedCapacity < memoryReservationStep)
94              {
95                  memory.ReservedCapacity = memoryReservationStep;
96              }
97              SetPointers(_memory);
98              // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
99              _memory.UsedCapacity = ((long)_header->AllocatedLinks * sizeof(Link)) +
                ↪  sizeof(LinksHeader);
100             // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
101             _header->ReservedLinks = (id)((_memory.ReservedCapacity - sizeof(LinksHeader)) /
                ↪  sizeof(Link));
102         }
103
104         [MethodImpl(MethodImplOptions.AggressiveInlining)]
105         public id Count(IList<id> restrictions)
106         {
107             // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
108             if (restrictions.Count == 0)
109             {
110                 return Total;
111             }
112             if (restrictions.Count == 1)
113             {
114                 var index = restrictions[Constants.IndexPart];
115                 if (index == Constants.Any)
116                 {
117                     return Total;
118                 }
119                 return Exists(index) ? 1UL : 0UL;
120             }
121             if (restrictions.Count == 2)
122             {
123                 var index = restrictions[Constants.IndexPart];
124                 var value = restrictions[1];
125                 if (index == Constants.Any)
126                 {
127                     if (value == Constants.Any)
128                     {
129                         return Total; // Any - как отсутствие ограничения
130                     }
131                     return _sourcesTreeMethods.CalculateReferences(value)
132                          + _targetsTreeMethods.CalculateReferences(value);
133                 }
134                 else
135                 {
136                     if (!Exists(index))
137                     {
138                         return 0;
139                     }
140                     if (value == Constants.Any)
141                     {
142                         return 1;
143                     }
144                     var storedLinkValue = GetLinkUnsafe(index);
145                     if (storedLinkValue->Source == value ||
146                         storedLinkValue->Target == value)
147                     {
148                         return 1;
149                     }
150                     return 0;
151                 }
152             }
153             if (restrictions.Count == 3)
154             {
155                 var index = restrictions[Constants.IndexPart];
156                 var source = restrictions[Constants.SourcePart];
157                 var target = restrictions[Constants.TargetPart];
158                 if (index == Constants.Any)
159                 {
160                     if (source == Constants.Any && target == Constants.Any)
161                     {
162                         return Total;
163                     }
164                     else if (source == Constants.Any)
165                     {
166                         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)
                    {
```

```
322                         if (storedLinkValue->Source == source &&
323                             storedLinkValue->Target == target)
324                         {
325                             return handler(GetLinkStruct(index));
326                         }
327                         return Constants.Continue;
328                     }
329                     var value = default(id);
330                     if (source == Constants.Any)
331                     {
332                         value = target;
333                     }
334                     if (target == Constants.Any)
335                     {
336                         value = source;
337                     }
338                     if (storedLinkValue->Source == value ||
339                         storedLinkValue->Target == value)
340                     {
341                         return handler(GetLinkStruct(index));
342                     }
343                     return Constants.Continue;
344                 }
345             }
346             throw new NotSupportedException("Другие размеры и способы ограничений не
                ↪  поддерживаются.");
347         }
348 
349         /// <remarks>
350         /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
                ↪  в другом месте (но не в менеджере памяти, а в логике Links)
351         /// </remarks>
352         [MethodImpl(MethodImplOptions.AggressiveInlining)]
353         public id Update(IList<id> values)
354         {
355             var linkIndex = values[Constants.IndexPart];
356             var link = GetLinkUnsafe(linkIndex);
357             // Будет корректно работать только в том случае, если пространство выделенной связи
                ↪  предварительно заполнено нулями
358             if (link->Source != Constants.Null)
359             {
360                 _sourcesTreeMethods.Detach(new IntPtr(&_header->FirstAsSource), linkIndex);
361             }
362             if (link->Target != Constants.Null)
363             {
364                 _targetsTreeMethods.Detach(new IntPtr(&_header->FirstAsTarget), linkIndex);
365             }
366 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
367             var leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
368             var rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
369             if (leftTreeSize != rightTreeSize)
370             {
371                 throw new Exception("One of the trees is broken.");
372             }
373 #endif
374             link->Source = values[Constants.SourcePart];
375             link->Target = values[Constants.TargetPart];
376             if (link->Source != Constants.Null)
377             {
378                 _sourcesTreeMethods.Attach(new IntPtr(&_header->FirstAsSource), linkIndex);
379             }
380             if (link->Target != Constants.Null)
381             {
382                 _targetsTreeMethods.Attach(new IntPtr(&_header->FirstAsTarget), linkIndex);
383             }
384 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
385             leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
386             rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
387             if (leftTreeSize != rightTreeSize)
388             {
389                 throw new Exception("One of the trees is broken.");
390             }
391 #endif
392             return linkIndex;
393         }
394 
395         [MethodImpl(MethodImplOptions.AggressiveInlining)]
396         private IList<id> GetLinkStruct(id linkIndex)
```

```
397         {
398             var link = GetLinkUnsafe(linkIndex);
399             return new UInt64Link(linkIndex, link->Source, link->Target);
400         }
401 
402         [MethodImpl(MethodImplOptions.AggressiveInlining)]
403         private Link* GetLinkUnsafe(id linkIndex) => &_links[linkIndex];
404 
405         /// <remarks>
406         /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
            ↪ пространство
407         /// </remarks>
408         public id Create()
409         {
410             var freeLink = _header->FirstFreeLink;
411             if (freeLink != Constants.Null)
412             {
413                 _unusedLinksListMethods.Detach(freeLink);
414             }
415             else
416             {
417                 if (_header->AllocatedLinks > Constants.MaxPossibleIndex)
418                 {
419                     throw new LinksLimitReachedException(Constants.MaxPossibleIndex);
420                 }
421                 if (_header->AllocatedLinks >= _header->ReservedLinks - 1)
422                 {
423                     _memory.ReservedCapacity += _memoryReservationStep;
424                     SetPointers(_memory);
425                     _header->ReservedLinks = (id)(_memory.ReservedCapacity / sizeof(Link));
426                 }
427                 _header->AllocatedLinks++;
428                 _memory.UsedCapacity += sizeof(Link);
429                 freeLink = _header->AllocatedLinks;
430             }
431             return freeLink;
432         }
433 
434         public void Delete(id link)
435         {
436             if (link < _header->AllocatedLinks)
437             {
438                 _unusedLinksListMethods.AttachAsFirst(link);
439             }
440             else if (link == _header->AllocatedLinks)
441             {
442                 _header->AllocatedLinks--;
443                 _memory.UsedCapacity -= sizeof(Link);
444                 // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
                    ↪ пока не дойдём до первой существующей связи
445                 // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
446                 while (_header->AllocatedLinks > 0 && IsUnusedLink(_header->AllocatedLinks))
447                 {
448                     _unusedLinksListMethods.Detach(_header->AllocatedLinks);
449                     _header->AllocatedLinks--;
450                     _memory.UsedCapacity -= sizeof(Link);
451                 }
452             }
453         }
454 
455         /// <remarks>
456         /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
            ↪ адрес реально поменялся
457         ///
458         /// Указатель this.links может быть в том же месте,
459         /// так как 0-я связь не используется и имеет такой же размер как Header,
460         /// поэтому header размещается в том же месте, что и 0-я связь
461         /// </remarks>
462         private void SetPointers(IResizableDirectMemory memory)
463         {
464             if (memory == null)
465             {
466                 _header = null;
467                 _links = null;
468                 _unusedLinksListMethods = null;
469                 _targetsTreeMethods = null;
470                 _unusedLinksListMethods = null;
471             }
472             else
```

```
            {
                _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

```
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

```
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)
        {
```

```
157             }
158
159             protected override IntPtr GetLeftPointer(ulong node) => new
                ↪  IntPtr(&Links[node].LeftAsSource);
160
161             protected override IntPtr GetRightPointer(ulong node) => new
                ↪  IntPtr(&Links[node].RightAsSource);
162
163             protected override ulong GetLeftValue(ulong node) => Links[node].LeftAsSource;
164
165             protected override ulong GetRightValue(ulong node) => Links[node].RightAsSource;
166
167             protected override ulong GetSize(ulong node)
168             {
169                 var previousValue = Links[node].SizeAsSource;
170                 //return MathHelpers.PartialRead(previousValue, 5, -5);
171                 return (previousValue & 4294967264) >> 5;
172             }
173
174             protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource
                ↪  = left;
175
176             protected override void SetRight(ulong node, ulong right) =>
                ↪  Links[node].RightAsSource = right;
177
178             protected override void SetSize(ulong node, ulong size)
179             {
180                 var previousValue = Links[node].SizeAsSource;
181                 //var modified = MathHelpers.PartialWrite(previousValue, size, 5, -5);
182                 var modified = (previousValue & 31) | ((size & 134217727) << 5);
183                 Links[node].SizeAsSource = modified;
184             }
185
186             protected override bool GetLeftIsChild(ulong node)
187             {
188                 var previousValue = Links[node].SizeAsSource;
189                 //return (Integer)MathHelpers.PartialRead(previousValue, 4, 1);
190                 return (previousValue & 16) >> 4 == 1UL;
191             }
192
193             protected override void SetLeftIsChild(ulong node, bool value)
194             {
195                 var previousValue = Links[node].SizeAsSource;
196                 //var modified = MathHelpers.PartialWrite(previousValue, (ulong)(Integer)value,
                    ↪  4, 1);
197                 var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
198                 Links[node].SizeAsSource = modified;
199             }
200
201             protected override bool GetRightIsChild(ulong node)
202             {
203                 var previousValue = Links[node].SizeAsSource;
204                 //return (Integer)MathHelpers.PartialRead(previousValue, 3, 1);
205                 return (previousValue & 8) >> 3 == 1UL;
206             }
207
208             protected override void SetRightIsChild(ulong node, bool value)
209             {
210                 var previousValue = Links[node].SizeAsSource;
211                 //var modified = MathHelpers.PartialWrite(previousValue, (ulong)(Integer)value,
                    ↪  3, 1);
212                 var modified = (previousValue & 4294967287) | ((value ? 1UL : 0UL) << 3);
213                 Links[node].SizeAsSource = modified;
214             }
215
216             protected override sbyte GetBalance(ulong node)
217             {
218                 var previousValue = Links[node].SizeAsSource;
219                 //var value = MathHelpers.PartialRead(previousValue, 0, 3);
220                 var value = previousValue & 7;
221                 var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
                    ↪  124 : value & 3);
222                 return unpackedValue;
223             }
224
225             protected override void SetBalance(ulong node, sbyte value)
226             {
227                 var previousValue = Links[node].SizeAsSource;
```

```
228                 var packagedValue = (ulong)((((byte)value >> 5) & 4) | value & 3);
229                 //var modified = MathHelpers.PartialWrite(previousValue, packagedValue, 0, 3);
230                 var modified = (previousValue & 4294967288) | (packagedValue & 7);
231                 Links[node].SizeAsSource = modified;
232             }
233 
234             protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
235                 => Links[first].Source < Links[second].Source ||
236                   (Links[first].Source == Links[second].Source && Links[first].Target <
                    ↪  Links[second].Target);
237 
238             protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
239                 => Links[first].Source > Links[second].Source ||
240                   (Links[first].Source == Links[second].Source && Links[first].Target >
                    ↪  Links[second].Target);
241 
242             protected override ulong GetTreeRoot() => Header->FirstAsSource;
243 
244             protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
245 
246             /// <summary>
247             /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
                ↪  (концом)
248             /// по дереву (индексу) связей, отсортированному по Source, а затем по Target.
249             /// </summary>
250             /// <param name="source">Индекс связи, которая является началом на искомой
                ↪  связи.</param>
251             /// <param name="target">Индекс связи, которая является концом на искомой
                ↪  связи.</param>
252             /// <returns>Индекс искомой связи.</returns>
253             public ulong Search(ulong source, ulong target)
254             {
255                 var root = Header->FirstAsSource;
256                 while (root != 0)
257                 {
258                     var rootSource = Links[root].Source;
259                     var rootTarget = Links[root].Target;
260                     if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                        ↪  node.Key < root.Key
261                     {
262                         root = GetLeftOrDefault(root);
263                     }
264                     else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget))
                        ↪  // node.Key > root.Key
265                     {
266                         root = GetRightOrDefault(root);
267                     }
268                     else // node.Key == root.Key
269                     {
270                         return root;
271                     }
272                 }
273                 return 0;
274             }
275 
276             [MethodImpl(MethodImplOptions.AggressiveInlining)]
277             private static bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
                ↪  ulong secondSource, ulong secondTarget)
278                 => firstSource < secondSource || (firstSource == secondSource && firstTarget <
                    ↪  secondTarget);
279 
280             [MethodImpl(MethodImplOptions.AggressiveInlining)]
281             private static bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
                ↪  ulong secondSource, ulong secondTarget)
282                 => firstSource > secondSource || (firstSource == secondSource && firstTarget >
                    ↪  secondTarget);
283 
284             [MethodImpl(MethodImplOptions.AggressiveInlining)]
285             protected override void ClearNode(ulong node)
286             {
287                 Links[node].LeftAsSource = 0UL;
288                 Links[node].RightAsSource = 0UL;
289                 Links[node].SizeAsSource = 0UL;
290             }
291 
292             [MethodImpl(MethodImplOptions.AggressiveInlining)]
293             protected override ulong GetZero() => 0UL;
294 
```

```
295             [MethodImpl(MethodImplOptions.AggressiveInlining)]
296             protected override ulong GetOne() => 1UL;
297
298             [MethodImpl(MethodImplOptions.AggressiveInlining)]
299             protected override ulong GetTwo() => 2UL;
300
301             [MethodImpl(MethodImplOptions.AggressiveInlining)]
302             protected override bool ValueEqualToZero(IntPtr pointer) =>
                ↪  *(ulong*)pointer.ToPointer() == 0UL;
303
304             [MethodImpl(MethodImplOptions.AggressiveInlining)]
305             protected override bool EqualToZero(ulong value) => value == 0UL;
306
307             [MethodImpl(MethodImplOptions.AggressiveInlining)]
308             protected override bool IsEquals(ulong first, ulong second) => first == second;
309
310             [MethodImpl(MethodImplOptions.AggressiveInlining)]
311             protected override bool GreaterThanZero(ulong value) => value > 0UL;
312
313             [MethodImpl(MethodImplOptions.AggressiveInlining)]
314             protected override bool GreaterThan(ulong first, ulong second) => first > second;
315
316             [MethodImpl(MethodImplOptions.AggressiveInlining)]
317             protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >=
                ↪  second;
318
319             [MethodImpl(MethodImplOptions.AggressiveInlining)]
320             protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0
                ↪  is always true for ulong
321
322             [MethodImpl(MethodImplOptions.AggressiveInlining)]
323             protected override bool LessOrEqualThanZero(ulong value) => value == 0; // value is
                ↪  always >= 0 for ulong
324
325             [MethodImpl(MethodImplOptions.AggressiveInlining)]
326             protected override bool LessOrEqualThan(ulong first, ulong second) => first <=
                ↪  second;
327
328             [MethodImpl(MethodImplOptions.AggressiveInlining)]
329             protected override bool LessThanZero(ulong value) => false; // value < 0 is always
                ↪  false for ulong
330
331             [MethodImpl(MethodImplOptions.AggressiveInlining)]
332             protected override bool LessThan(ulong first, ulong second) => first < second;
333
334             [MethodImpl(MethodImplOptions.AggressiveInlining)]
335             protected override ulong Increment(ulong value) => ++value;
336
337             [MethodImpl(MethodImplOptions.AggressiveInlining)]
338             protected override ulong Decrement(ulong value) => --value;
339
340             [MethodImpl(MethodImplOptions.AggressiveInlining)]
341             protected override ulong Add(ulong first, ulong second) => first + second;
342
343             [MethodImpl(MethodImplOptions.AggressiveInlining)]
344             protected override ulong Subtract(ulong first, ulong second) => first - second;
345         }
346
347         private class LinksTargetsTreeMethods : LinksTreeMethodsBase
348         {
349             public LinksTargetsTreeMethods(UInt64ResizableDirectMemoryLinks memory)
350                 : base(memory)
351             {
352             }
353
354             //protected override IntPtr GetLeft(ulong node) => new
                ↪  IntPtr(&Links[node].LeftAsTarget);
355
356             //protected override IntPtr GetRight(ulong node) => new
                ↪  IntPtr(&Links[node].RightAsTarget);
357
358             //protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;
359
360             //protected override void SetLeft(ulong node, ulong left) =>
                ↪  Links[node].LeftAsTarget = left;
361
362             //protected override void SetRight(ulong node, ulong right) =>
                ↪  Links[node].RightAsTarget = right;
363
```

```
364             //protected override void SetSize(ulong node, ulong size) =>
                ↪  Links[node].SizeAsTarget = size;
365
366             protected override IntPtr GetLeftPointer(ulong node) => new
                ↪  IntPtr(&Links[node].LeftAsTarget);
367
368             protected override IntPtr GetRightPointer(ulong node) => new
                ↪  IntPtr(&Links[node].RightAsTarget);
369
370             protected override ulong GetLeftValue(ulong node) => Links[node].LeftAsTarget;
371
372             protected override ulong GetRightValue(ulong node) => Links[node].RightAsTarget;
373
374             protected override ulong GetSize(ulong node)
375             {
376                 var previousValue = Links[node].SizeAsTarget;
377                 //return MathHelpers.PartialRead(previousValue, 5, -5);
378                 return (previousValue & 4294967264) >> 5;
379             }
380
381             protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget
                ↪  = left;
382
383             protected override void SetRight(ulong node, ulong right) =>
                ↪  Links[node].RightAsTarget = right;
384
385             protected override void SetSize(ulong node, ulong size)
386             {
387                 var previousValue = Links[node].SizeAsTarget;
388                 //var modified = MathHelpers.PartialWrite(previousValue, size, 5, -5);
389                 var modified = (previousValue & 31) | ((size & 134217727) << 5);
390                 Links[node].SizeAsTarget = modified;
391             }
392
393             protected override bool GetLeftIsChild(ulong node)
394             {
395                 var previousValue = Links[node].SizeAsTarget;
396                 //return (Integer)MathHelpers.PartialRead(previousValue, 4, 1);
397                 return (previousValue & 16) >> 4 == 1UL;
398                 // TODO: Check if this is possible to use
399                 //var nodeSize = GetSize(node);
400                 //var left = GetLeftValue(node);
401                 //var leftSize = GetSizeOrZero(left);
402                 //return leftSize > 0 && nodeSize > leftSize;
403             }
404
405             protected override void SetLeftIsChild(ulong node, bool value)
406             {
407                 var previousValue = Links[node].SizeAsTarget;
408                 //var modified = MathHelpers.PartialWrite(previousValue, (ulong)(Integer)value,
                    ↪  4, 1);
409                 var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
410                 Links[node].SizeAsTarget = modified;
411             }
412
413             protected override bool GetRightIsChild(ulong node)
414             {
415                 var previousValue = Links[node].SizeAsTarget;
416                 //return (Integer)MathHelpers.PartialRead(previousValue, 3, 1);
417                 return (previousValue & 8) >> 3 == 1UL;
418                 // TODO: Check if this is possible to use
419                 //var nodeSize = GetSize(node);
420                 //var right = GetRightValue(node);
421                 //var rightSize = GetSizeOrZero(right);
422                 //return rightSize > 0 && nodeSize > rightSize;
423             }
424
425             protected override void SetRightIsChild(ulong node, bool value)
426             {
427                 var previousValue = Links[node].SizeAsTarget;
428                 //var modified = MathHelpers.PartialWrite(previousValue, (ulong)(Integer)value,
                    ↪  3, 1);
429                 var modified = (previousValue & 4294967287) | ((value ? 1UL : 0UL) << 3);
430                 Links[node].SizeAsTarget = modified;
431             }
432
433             protected override sbyte GetBalance(ulong node)
434             {
```

```
                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 = OUL;
                Links[node].RightAsTarget = OUL;
                Links[node].SizeAsTarget = OUL;
            }
        }
    }
}
```

./Sequences/Converters/BalancedVariantConverter.cs

```
using System.Collections.Generic;

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

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

```
        }

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

./Sequences/Converters/CompressingConverter.cs

```
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)
        {
        }
```

```
54
55          public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
            ↪  baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
            ↪  minFrequencyToCompress, bool doInitialFrequenciesIncrement)
56              : base(links)
57          {
58              _baseConverter = baseConverter;
59              _doubletFrequenciesCache = doubletFrequenciesCache;
60              if (_comparer.Compare(minFrequencyToCompress, Integer<TLink>.One) < 0)
61              {
62                  minFrequencyToCompress = Integer<TLink>.One;
63              }
64              _minFrequencyToCompress = minFrequencyToCompress;
65              _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
66              ResetMaxDoublet();
67          }
68
69          public override TLink Convert(IList<TLink> source) =>
            ↪  _baseConverter.Convert(Compress(source));
70
71          /// <remarks>
72          /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding .
73          /// Faster version (doublets' frequencies dictionary is not recreated).
74          /// </remarks>
75          private IList<TLink> Compress(IList<TLink> sequence)
76          {
77              if (sequence.IsNullOrEmpty())
78              {
79                  return null;
80              }
81              if (sequence.Count == 1)
82              {
83                  return sequence;
84              }
85              if (sequence.Count == 2)
86              {
87                  return new[] { Links.GetOrCreate(sequence[0], sequence[1]) };
88              }
89              // TODO: arraypool with min size (to improve cache locality) or stackallow with Sigil
90              var copy = new HalfDoublet[sequence.Count];
91              Doublet<TLink> doublet = default;
92              for (var i = 1; i < sequence.Count; i++)
93              {
94                  doublet.Source = sequence[i - 1];
95                  doublet.Target = sequence[i];
96                  LinkFrequency<TLink> data;
97                  if (_doInitialFrequenciesIncrement)
98                  {
99                      data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
100                 }
101                 else
102                 {
103                     data = _doubletFrequenciesCache.GetFrequency(ref doublet);
104                     if (data == null)
105                     {
106                         throw new NotSupportedException("If you ask not to increment
                            ↪  frequencies, it is expected that all frequencies for the sequence
                            ↪  are prepared.");
107                     }
108                 }
109                 copy[i - 1].Element = sequence[i - 1];
110                 copy[i - 1].DoubletData = data;
111                 UpdateMaxDoublet(ref doublet, data);
112             }
113             copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
114             copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
115             if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
116             {
117                 var newLength = ReplaceDoublets(copy);
118                 sequence = new TLink[newLength];
119                 for (int i = 0; i < newLength; i++)
120                 {
121                     sequence[i] = copy[i].Element;
122                 }
123             }
124             return sequence;
125         }
126
127         /// <remarks>
```

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

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

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

```

```
202         [MethodImpl(MethodImplOptions.AggressiveInlining)]
203         private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
204         {
205             var frequency = data.Frequency;
206             var maxFrequency = _maxDoubletData.Frequency;
207             //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
                ↪  (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
                ↪  compression string data (and gives collisions quickly) */ _maxDoublet.Source +
                ↪  _maxDoublet.Target)))
208             if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
209                (_comparer.Compare(maxFrequency, frequency) < 0 ||
                    ↪  (_equalityComparer.Equals(maxFrequency, frequency) &&
                    ↪  _comparer.Compare(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) */
210             {
211                 _maxDoublet = doublet;
212                 _maxDoubletData = data;
213             }
214         }
215     }
216 }
```

./Sequences/Converters/LinksListToSequenceConverterBase.cs

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

./Sequences/Converters/OptimalVariantConverter.cs

```
1  using System.Collections.Generic;
2  using System.Linq;
3  using Platform.Interfaces;
4
5  namespace Platform.Data.Doublets.Sequences.Converters
6  {
7      public class OptimalVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
8      {
9          private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
10         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
11
12         private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;
13
14         public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
           ↪  sequenceToItsLocalElementLevelsConverter) : base(links)
15             => _sequenceToItsLocalElementLevelsConverter =
                ↪  sequenceToItsLocalElementLevelsConverter;
16
17         public override TLink Convert(IList<TLink> sequence)
18         {
19             var length = sequence.Count;
20             if (length == 1)
21             {
22                 return sequence[0];
23             }
24             var links = Links;
25             if (length == 2)
26             {
27                 return links.GetOrCreate(sequence[0], sequence[1]);
28             }
29             sequence = sequence.ToArray();
30             var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
31             while (length > 2)
32             {
33                 var levelRepeat = 1;
34                 var currentLevel = levels[0];
35                 var previousLevel = levels[0];
36                 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

```
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

```
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

```
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

```
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

```
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

```
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
{
```

```
16      public class DuplicateSegmentsProvider<TLink> :
        ↪  DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
        ↪  IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
17      {
18          private readonly ILinks<TLink> _links;
19          private readonly ISequences<TLink> _sequences;
20          private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
21          private BitString _visited;
22
23          private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
            ↪  IList<TLink>>>
24          {
25              private readonly IListEqualityComparer<TLink> _listComparer;
26              public ItemEquilityComparer() => _listComparer =
                ↪  Default<IListEqualityComparer<TLink>>.Instance;
27              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);
28              public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
                ↪  HashHelpers.Generate(_listComparer.GetHashCode(pair.Key),
                ↪  _listComparer.GetHashCode(pair.Value));
29          }
30
31          private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
32          {
33              private readonly IListComparer<TLink> _listComparer;
34
35              public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;
36
37              public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
                ↪  KeyValuePair<IList<TLink>, IList<TLink>> right)
38              {
39                  var intermediateResult = _listComparer.Compare(left.Key, right.Key);
40                  if (intermediateResult == 0)
41                  {
42                      intermediateResult = _listComparer.Compare(left.Value, right.Value);
43                  }
44                  return intermediateResult;
45              }
46          }
47
48          public DuplicateSegmentsProvider(ILinks<TLink> links, ISequences<TLink> sequences)
49              : base(minimumStringSegmentLength: 2)
50          {
51              _links = links;
52              _sequences = sequences;
53          }
54
55          public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
56          {
57              _groups = new HashSet<KeyValuePair<IList<TLink>,
                ↪  IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
58              var count = _links.Count();
59              _visited = new BitString((long)(Integer<TLink>)count + 1);
60              _links.Each(link =>
61              {
62                  var linkIndex = _links.GetIndex(link);
63                  var linkBitIndex = (long)(Integer<TLink>)linkIndex;
64                  if (!_visited.Get(linkBitIndex))
65                  {
66                      var sequenceElements = new List<TLink>();
67                      _sequences.EachPart(sequenceElements.AddAndReturnTrue, linkIndex);
68                      if (sequenceElements.Count > 2)
69                      {
70                          WalkAll(sequenceElements);
71                      }
72                  }
73                  return _links.Constants.Continue;
74              });
75              var resultList = _groups.ToList();
76              var comparer = Default<ItemComparer>.Instance;
77              resultList.Sort(comparer);
78  #if DEBUG
79              foreach (var item in resultList)
80              {
81                  PrintDuplicates(item);
82              }
83  #endif
```

```
84              return resultList;
85          }
86
87          protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
            ↪  length) => new Segment<TLink>(elements, offset, length);
88
89          protected override void OnDublicateFound(Segment<TLink> segment)
90          {
91              var duplicates = CollectDuplicatesForSegment(segment);
92              if (duplicates.Count > 1)
93              {
94                  _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
                    ↪  duplicates));
95              }
96          }
97
98          private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
99          {
100             var duplicates = new List<TLink>();
101             var readAsElement = new HashSet<TLink>();
102             _sequences.Each(sequence =>
103             {
104                 duplicates.Add(sequence);
105                 readAsElement.Add(sequence);
106                 return true; // Continue
107             }, segment);
108             if (duplicates.Any(x => _visited.Get((Integer<TLink>)x)))
109             {
110                 return new List<TLink>();
111             }
112             foreach (var duplicate in duplicates)
113             {
114                 var duplicateBitIndex = (long)(Integer<TLink>)duplicate;
115                 _visited.Set(duplicateBitIndex);
116             }
117             if (_sequences is Sequences sequencesExperiments)
118             {
119                 var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((H
                    ↪  ashSet<ulong>)(object)readAsElement,
                    ↪  (IList<ulong>)segment);
120                 foreach (var partiallyMatchedSequence in partiallyMatched)
121                 {
122                     TLink sequenceIndex = (Integer<TLink>)partiallyMatchedSequence;
123                     duplicates.Add(sequenceIndex);
124                 }
125             }
126             duplicates.Sort();
127             return duplicates;
128         }
129
130         private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
131         {
132             if (!(_links is ILinks<ulong> ulongLinks))
133             {
134                 return;
135             }
136             var duplicatesKey = duplicatesItem.Key;
137             var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
138             Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)})");
139             var duplicatesList = duplicatesItem.Value;
140             for (int i = 0; i < duplicatesList.Count; i++)
141             {
142                 ulong sequenceIndex = (Integer<TLink>)duplicatesList[i];
143                 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));
144                 Console.WriteLine(formatedSequenceStructure);
145                 var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
                    ↪  ulongLinks);
146                 Console.WriteLine(sequenceString);
147             }
148             Console.WriteLine();
149         }
150     }
151 }
```

./Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkFrequencyIncrementer.cs

```
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/FrequenciesCacheBasedLinkToItsFrequencyNumberConverter.cs

```
using Platform.Interfaces;

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

./Sequences/Frequencies/Cache/LinkFrequenciesCache.cs

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
36          public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
37          {
38              _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
39              return data;
40          }
41
42          public void IncrementFrequencies(IList<TLink> sequence)
43          {
44              for (var i = 1; i < sequence.Count; i++)
45              {
46                  IncrementFrequency(sequence[i - 1], sequence[i]);
47              }
48          }
49
50          [MethodImpl(MethodImplOptions.AggressiveInlining)]
51          public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
52          {
53              var doublet = new Doublet<TLink>(source, target);
54              return IncrementFrequency(ref doublet);
55          }
56
57          public void PrintFrequencies(IList<TLink> sequence)
58          {
59              for (var i = 1; i < sequence.Count; i++)
60              {
61                  PrintFrequency(sequence[i - 1], sequence[i]);
62              }
63          }
64
65          public void PrintFrequency(TLink source, TLink target)
66          {
67              var number = GetFrequency(source, target).Frequency;
68              Console.WriteLine("({0},{1}) - {2}", source, target, number);
69          }
70
71          [MethodImpl(MethodImplOptions.AggressiveInlining)]
72          public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
73          {
74              if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
75              {
76                  data.IncrementFrequency();
77              }
78              else
79              {
80                  var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
81                  data = new LinkFrequency<TLink>(Integer<TLink>.One, link);
82                  if (!_equalityComparer.Equals(link, default))
83                  {
84                      data.Frequency = ArithmeticHelpers.Add(data.Frequency,
                        ↪  _frequencyCounter.Count(link));
85                  }
86                  _doubletsCache.Add(doublet, data);
87              }
88              return data;
89          }
90
91          public void ValidateFrequencies()
92          {
93              foreach (var entry in _doubletsCache)
94              {
95                  var value = entry.Value;
96                  var linkIndex = value.Link;
97                  if (!_equalityComparer.Equals(linkIndex, default))
98                  {
99                      var frequency = value.Frequency;
100                     var count = _frequencyCounter.Count(linkIndex);
101                     // TODO: Why `frequency` always greater than `count` by 1?
102                     if (((_comparer.Compare(frequency, count) > 0) &&
                        ↪  (_comparer.Compare(ArithmeticHelpers.Subtract(frequency, count),
                        ↪  Integer<TLink>.One) > 0))
103                      || ((_comparer.Compare(count, frequency) > 0) &&
                         ↪  (_comparer.Compare(ArithmeticHelpers.Subtract(count, frequency),
                         ↪  Integer<TLink>.One) > 0)))
104                     {
105                         throw new InvalidOperationException("Frequencies validation failed.");
106                     }
107                 }
108                 //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

```
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

```
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

```
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

```
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.cs

```
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

```
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

```
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);
            }
        }

```

```
48          protected virtual void CountSequenceSymbolFrequency(TLink link)
49          {
50              var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
                ↪  link, _symbol);
51              _total = ArithmeticHelpers.Add(_total, symbolFrequencyCounter.Count());
52          }
53
54          private TLink EachElementHandler(IList<TLink> doublet)
55          {
56              var constants = _links.Constants;
57              var doubletIndex = doublet[constants.IndexPart];
58              if (_visits.Add(doubletIndex))
59              {
60                  CountCore(doubletIndex);
61              }
62              return constants.Continue;
63          }
64      }
65  }
```

./Sequences/HeightProviders/CachedSequenceHeightProvider.cs

```
1   using System.Collections.Generic;
2   using Platform.Interfaces;
3
4   namespace Platform.Data.Doublets.Sequences.HeightProviders
5   {
6       public class CachedSequenceHeightProvider<TLink> : LinksOperatorBase<TLink>,
        ↪  ISequenceHeightProvider<TLink>
7       {
8           private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
9
10          private readonly TLink _heightPropertyMarker;
11          private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
12          private readonly IConverter<TLink> _addressToUnaryNumberConverter;
13          private readonly IConverter<TLink> _unaryNumberToAddressConverter;
14          private readonly IPropertyOperator<TLink, TLink, TLink> _propertyOperator;
15
16          public CachedSequenceHeightProvider(
17              ILinks<TLink> links,
18              ISequenceHeightProvider<TLink> baseHeightProvider,
19              IConverter<TLink> addressToUnaryNumberConverter,
20              IConverter<TLink> unaryNumberToAddressConverter,
21              TLink heightPropertyMarker,
22              IPropertyOperator<TLink, TLink, TLink> propertyOperator)
23              : base(links)
24          {
25              _heightPropertyMarker = heightPropertyMarker;
26              _baseHeightProvider = baseHeightProvider;
27              _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
28              _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
29              _propertyOperator = propertyOperator;
30          }
31
32          public TLink Get(TLink sequence)
33          {
34              TLink height;
35              var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
36              if (_equalityComparer.Equals(heightValue, default))
37              {
38                  height = _baseHeightProvider.Get(sequence);
39                  heightValue = _addressToUnaryNumberConverter.Convert(height);
40                  _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
41              }
42              else
43              {
44                  height = _unaryNumberToAddressConverter.Convert(heightValue);
45              }
46              return height;
47          }
48      }
49  }
```

./Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs

```
1   using Platform.Interfaces;
2   using Platform.Numbers;
3
4   namespace Platform.Data.Doublets.Sequences.HeightProviders
5   {
6       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

```
using Platform.Interfaces;

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

./Sequences/Sequences.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.CompilerServices;
using Platform.Collections;
using Platform.Collections.Lists;
using Platform.Threading.Synchronization;
using Platform.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.
    ///
```

```
44      /// Можно ли как-то сделать один общий интерфейс
45      ///
46      ///
47      /// Блокчейн и/или гит для распределённой записи транзакций.
48      ///
49      /// </remarks>
50      public partial class Sequences : ISequences<ulong> // IList<string>, IList<ulong[]> (после
        ↪  завершения реализации Sequences)
51      {
52          private static readonly LinksCombinedConstants<bool, ulong, long> _constants =
            ↪  Default<LinksCombinedConstants<bool, ulong, long>>.Instance;
53
54          /// <summary>Возвращает значение ulong, обозначающее любое количество связей.</summary>
55          public const ulong ZeroOrMany = ulong.MaxValue;
56
57          public SequencesOptions<ulong> Options;
58          public readonly SynchronizedLinks<ulong> Links;
59          public readonly ISynchronization Sync;
60
61          public Sequences(SynchronizedLinks<ulong> links)
62              : this(links, new SequencesOptions<ulong>())
63          {
64          }
65
66          public Sequences(SynchronizedLinks<ulong> links, SequencesOptions<ulong> options)
67          {
68              Links = links;
69              Sync = links.SyncRoot;
70              Options = options;
71
72              Options.ValidateOptions();
73              Options.InitOptions(Links);
74          }
75
76          public bool IsSequence(ulong sequence)
77          {
78              return Sync.ExecuteReadOperation(() =>
79              {
80                  if (Options.UseSequenceMarker)
81                  {
82                      return Options.MarkedSequenceMatcher.IsMatched(sequence);
83                  }
84                  return !Links.Unsync.IsPartialPoint(sequence);
85              });
86          }
87
88          [MethodImpl(MethodImplOptions.AggressiveInlining)]
89          private ulong GetSequenceByElements(ulong sequence)
90          {
91              if (Options.UseSequenceMarker)
92              {
93                  return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
94              }
95              return sequence;
96          }
97
98          private ulong GetSequenceElements(ulong sequence)
99          {
100             if (Options.UseSequenceMarker)
101             {
102                 var linkContents = new UInt64Link(Links.GetLink(sequence));
103                 if (linkContents.Source == Options.SequenceMarkerLink)
104                 {
105                     return linkContents.Target;
106                 }
107                 if (linkContents.Target == Options.SequenceMarkerLink)
108                 {
109                     return linkContents.Source;
110                 }
111             }
112             return sequence;
113         }
114
115         #region Count
116
117         public ulong Count(params ulong[] sequence)
118         {
119             if (sequence.Length == 0)
120             {
```

```
                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)
            {
```

```
277                 if (!StepLeft(innerHandler, sequence[sequence.Count - 2],
                    ↪ sequence[sequence.Count - 1]))
278                 {
279                     return false;
280                 }
281             }
282             return true;
283         }
284 
285         private bool PartialStepRight(Func<ulong, bool> handler, ulong left, ulong right)
286         {
287             return Links.Unsync.Each(_constants.Any, left, doublet =>
288             {
289                 if (!StepRight(handler, doublet, right))
290                 {
291                     return false;
292                 }
293                 if (left != doublet)
294                 {
295                     return PartialStepRight(handler, doublet, right);
296                 }
297                 return true;
298             });
299         }
300 
301         private bool StepRight(Func<ulong, bool> handler, ulong left, ulong right) =>
                ↪ Links.Unsync.Each(left, _constants.Any, rightStep => TryStepRightUp(handler, right,
                ↪ rightStep));
302 
303         private bool TryStepRightUp(Func<ulong, bool> handler, ulong right, ulong stepFrom)
304         {
305             var upStep = stepFrom;
306             var firstSource = Links.Unsync.GetTarget(upStep);
307             while (firstSource != right && firstSource != upStep)
308             {
309                 upStep = firstSource;
310                 firstSource = Links.Unsync.GetSource(upStep);
311             }
312             if (firstSource == right)
313             {
314                 return handler(stepFrom);
315             }
316             return true;
317         }
318 
319         private bool StepLeft(Func<ulong, bool> handler, ulong left, ulong right) =>
                ↪ Links.Unsync.Each(_constants.Any, right, leftStep => TryStepLeftUp(handler, left,
                ↪ leftStep));
320 
321         private bool TryStepLeftUp(Func<ulong, bool> handler, ulong left, ulong stepFrom)
322         {
323             var upStep = stepFrom;
324             var firstTarget = Links.Unsync.GetSource(upStep);
325             while (firstTarget != left && firstTarget != upStep)
326             {
327                 upStep = firstTarget;
328                 firstTarget = Links.Unsync.GetTarget(upStep);
329             }
330             if (firstTarget == left)
331             {
332                 return handler(stepFrom);
333             }
334             return true;
335         }
336 
337         #endregion
338 
339         #region Update
340 
341         public ulong Update(ulong[] sequence, ulong[] newSequence)
342         {
343             if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
344             {
345                 return _constants.Null;
346             }
347             if (sequence.IsNullOrEmpty())
348             {
349                 return Create(newSequence);
350             }
```

```
351             if (newSequence.IsNullOrEmpty())
352             {
353                 Delete(sequence);
354                 return _constants.Null;
355             }
356             return Sync.ExecuteWriteOperation(() =>
357             {
358                 Links.EnsureEachLinkIsAnyOrExists(sequence);
359                 Links.EnsureEachLinkExists(newSequence);
360                 return UpdateCore(sequence, newSequence);
361             });
362         }
363 
364         private ulong UpdateCore(ulong[] sequence, ulong[] newSequence)
365         {
366             ulong bestVariant;
367             if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
                ↪  !sequence.EqualTo(newSequence))
368             {
369                 bestVariant = CompactCore(newSequence);
370             }
371             else
372             {
373                 bestVariant = CreateCore(newSequence);
374             }
375             // TODO: Check all options only ones before loop execution
376             // Возможно нужно две версии Each, возвращающий фактические последовательности и с
                ↪  маркером,
377             // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
                ↪  можно получить имея только фактические последовательности.
378             foreach (var variant in Each(sequence))
379             {
380                 if (variant != bestVariant)
381                 {
382                     UpdateOneCore(variant, bestVariant);
383                 }
384             }
385             return bestVariant;
386         }
387 
388         private void UpdateOneCore(ulong sequence, ulong newSequence)
389         {
390             if (Options.UseGarbageCollection)
391             {
392                 var sequenceElements = GetSequenceElements(sequence);
393                 var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
394                 var sequenceLink = GetSequenceByElements(sequenceElements);
395                 var newSequenceElements = GetSequenceElements(newSequence);
396                 var newSequenceLink = GetSequenceByElements(newSequenceElements);
397                 if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
398                 {
399                     if (sequenceLink != _constants.Null)
400                     {
401                         Links.Unsync.Merge(sequenceLink, newSequenceLink);
402                     }
403                     Links.Unsync.Merge(sequenceElements, newSequenceElements);
404                 }
405                 ClearGarbage(sequenceElementsContents.Source);
406                 ClearGarbage(sequenceElementsContents.Target);
407             }
408             else
409             {
410                 if (Options.UseSequenceMarker)
411                 {
412                     var sequenceElements = GetSequenceElements(sequence);
413                     var sequenceLink = GetSequenceByElements(sequenceElements);
414                     var newSequenceElements = GetSequenceElements(newSequence);
415                     var newSequenceLink = GetSequenceByElements(newSequenceElements);
416                     if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
417                     {
418                         if (sequenceLink != _constants.Null)
419                         {
420                             Links.Unsync.Merge(sequenceLink, newSequenceLink);
421                         }
422                         Links.Unsync.Merge(sequenceElements, newSequenceElements);
423                     }
424                 }
425                 else
```

```
426                 {
427                     if (Options.UseCascadeUpdate || CountReferences(sequence) == 0)
428                     {
429                         Links.Unsync.Merge(sequence, newSequence);
430                     }
431                 }
432             }
433         }
434 
435         #endregion
436 
437         #region Delete
438 
439         public void Delete(params ulong[] sequence)
440         {
441             Sync.ExecuteWriteOperation(() =>
442             {
443                 // TODO: Check all options only ones before loop execution
444                 foreach (var linkToDelete in Each(sequence))
445                 {
446                     DeleteOneCore(linkToDelete);
447                 }
448             });
449         }
450 
451         private void DeleteOneCore(ulong link)
452         {
453             if (Options.UseGarbageCollection)
454             {
455                 var sequenceElements = GetSequenceElements(link);
456                 var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
457                 var sequenceLink = GetSequenceByElements(sequenceElements);
458                 if (Options.UseCascadeDelete || CountReferences(link) == 0)
459                 {
460                     if (sequenceLink != _constants.Null)
461                     {
462                         Links.Unsync.Delete(sequenceLink);
463                     }
464                     Links.Unsync.Delete(link);
465                 }
466                 ClearGarbage(sequenceElementsContents.Source);
467                 ClearGarbage(sequenceElementsContents.Target);
468             }
469             else
470             {
471                 if (Options.UseSequenceMarker)
472                 {
473                     var sequenceElements = GetSequenceElements(link);
474                     var sequenceLink = GetSequenceByElements(sequenceElements);
475                     if (Options.UseCascadeDelete || CountReferences(link) == 0)
476                     {
477                         if (sequenceLink != _constants.Null)
478                         {
479                             Links.Unsync.Delete(sequenceLink);
480                         }
481                         Links.Unsync.Delete(link);
482                     }
483                 }
484                 else
485                 {
486                     if (Options.UseCascadeDelete || CountReferences(link) == 0)
487                     {
488                         Links.Unsync.Delete(link);
489                     }
490                 }
491             }
492         }
493 
494         #endregion
495 
496         #region Compactification
497 
498         /// <remarks>
499         /// bestVariant можно выбирать по максимальному числу использований,
500         /// но балансированный позволяет гарантировать уникальность (если есть возможность,
501         /// гарантировать его использование в других местах).
502         ///
503         /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
504         /// </remarks>
```

```
505             public ulong Compact(params ulong[] sequence)
506             {
507                 return Sync.ExecuteWriteOperation(() =>
508                 {
509                     if (sequence.IsNullOrEmpty())
510                     {
511                         return _constants.Null;
512                     }
513                     Links.EnsureEachLinkExists(sequence);
514                     return CompactCore(sequence);
515                 });
516             }
517 
518             [MethodImpl(MethodImplOptions.AggressiveInlining)]
519             private ulong CompactCore(params ulong[] sequence) => UpdateCore(sequence, sequence);
520 
521             #endregion
522 
523             #region Garbage Collection
524 
525             /// <remarks>
526             /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
                 ↪  определить извне или в унаследованном классе
527             /// </remarks>
528             [MethodImpl(MethodImplOptions.AggressiveInlining)]
529             private bool IsGarbage(ulong link) => link != Options.SequenceMarkerLink &&
                 ↪  !Links.Unsync.IsPartialPoint(link) && Links.Count(link) == 0;
530 
531             private void ClearGarbage(ulong link)
532             {
533                 if (IsGarbage(link))
534                 {
535                     var contents = new UInt64Link(Links.GetLink(link));
536                     Links.Unsync.Delete(link);
537                     ClearGarbage(contents.Source);
538                     ClearGarbage(contents.Target);
539                 }
540             }
541 
542             #endregion
543 
544             #region Walkers
545 
546             public bool EachPart(Func<ulong, bool> handler, ulong sequence)
547             {
548                 return Sync.ExecuteReadOperation(() =>
549                 {
550                     var links = Links.Unsync;
551                     var walker = new RightSequenceWalker<ulong>(links);
552                     foreach (var part in walker.Walk(sequence))
553                     {
554                         if (!handler(links.GetIndex(part)))
555                         {
556                             return false;
557                         }
558                     }
559                     return true;
560                 });
561             }
562 
563             public class Matcher : RightSequenceWalker<ulong>
564             {
565                 private readonly Sequences _sequences;
566                 private readonly IList<LinkIndex> _patternSequence;
567                 private readonly HashSet<LinkIndex> _linksInSequence;
568                 private readonly HashSet<LinkIndex> _results;
569                 private readonly Func<ulong, bool> _stopableHandler;
570                 private readonly HashSet<ulong> _readAsElements;
571                 private int _filterPosition;
572 
573                 public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
                     ↪  HashSet<LinkIndex> results, Func<LinkIndex, bool> stopableHandler,
                     ↪  HashSet<LinkIndex> readAsElements = null)
574                     : base(sequences.Links.Unsync)
575                 {
576                     _sequences = sequences;
577                     _patternSequence = patternSequence;
578                     _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                         ↪  _constants.Any && x != ZeroOrMany));
579                     _results = results;
```

```
580                 _stopableHandler = stopableHandler;
581                 _readAsElements = readAsElements;
582             }
583 
584             protected override bool IsElement(IList<ulong> link) => base.IsElement(link) ||
                ↪  (_readAsElements != null && _readAsElements.Contains(Links.GetIndex(link))) ||
                ↪  _linksInSequence.Contains(Links.GetIndex(link));
585 
586             public bool FullMatch(LinkIndex sequenceToMatch)
587             {
588                 _filterPosition = 0;
589                 foreach (var part in Walk(sequenceToMatch))
590                 {
591                     if (!FullMatchCore(Links.GetIndex(part)))
592                     {
593                         break;
594                     }
595                 }
596                 return _filterPosition == _patternSequence.Count;
597             }
598 
599             private bool FullMatchCore(LinkIndex element)
600             {
601                 if (_filterPosition == _patternSequence.Count)
602                 {
603                     _filterPosition = -2; // Длиннее чем нужно
604                     return false;
605                 }
606                 if (_patternSequence[_filterPosition] != _constants.Any
607                  && element != _patternSequence[_filterPosition])
608                 {
609                     _filterPosition = -1;
610                     return false; // Начинается/Продолжается иначе
611                 }
612                 _filterPosition++;
613                 return true;
614             }
615 
616             public void AddFullMatchedToResults(ulong sequenceToMatch)
617             {
618                 if (FullMatch(sequenceToMatch))
619                 {
620                     _results.Add(sequenceToMatch);
621                 }
622             }
623 
624             public bool HandleFullMatched(ulong sequenceToMatch)
625             {
626                 if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
627                 {
628                     return _stopableHandler(sequenceToMatch);
629                 }
630                 return true;
631             }
632 
633             public bool HandleFullMatchedSequence(ulong sequenceToMatch)
634             {
635                 var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
636                 if (sequence != _constants.Null && FullMatch(sequenceToMatch) &&
                    ↪  _results.Add(sequenceToMatch))
637                 {
638                     return _stopableHandler(sequence);
639                 }
640                 return true;
641             }
642 
643             /// <remarks>
644             /// TODO: Add support for LinksConstants.Any
645             /// </remarks>
646             public bool PartialMatch(LinkIndex sequenceToMatch)
647             {
648                 _filterPosition = -1;
649                 foreach (var part in Walk(sequenceToMatch))
650                 {
651                     if (!PartialMatchCore(Links.GetIndex(part)))
652                     {
653                         break;
654                     }
655                 }
```

```
656                 return _filterPosition == _patternSequence.Count - 1;
657             }
658
659             private bool PartialMatchCore(LinkIndex element)
660             {
661                 if (_filterPosition == (_patternSequence.Count - 1))
662                 {
663                     return false; // Нашлось
664                 }
665                 if (_filterPosition >= 0)
666                 {
667                     if (element == _patternSequence[_filterPosition + 1])
668                     {
669                         _filterPosition++;
670                     }
671                     else
672                     {
673                         _filterPosition = -1;
674                     }
675                 }
676                 if (_filterPosition < 0)
677                 {
678                     if (element == _patternSequence[0])
679                     {
680                         _filterPosition = 0;
681                     }
682                 }
683                 return true; // Ищем дальше
684             }
685
686             public void AddPartialMatchedToResults(ulong sequenceToMatch)
687             {
688                 if (PartialMatch(sequenceToMatch))
689                 {
690                     _results.Add(sequenceToMatch);
691                 }
692             }
693
694             public bool HandlePartialMatched(ulong sequenceToMatch)
695             {
696                 if (PartialMatch(sequenceToMatch))
697                 {
698                     return _stopableHandler(sequenceToMatch);
699                 }
700                 return true;
701             }
702
703             public void AddAllPartialMatchedToResults(IEnumerable<ulong> sequencesToMatch)
704             {
705                 foreach (var sequenceToMatch in sequencesToMatch)
706                 {
707                     if (PartialMatch(sequenceToMatch))
708                     {
709                         _results.Add(sequenceToMatch);
710                     }
711                 }
712             }
713
714             public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<ulong>
                ↪ sequencesToMatch)
715             {
716                 foreach (var sequenceToMatch in sequencesToMatch)
717                 {
718                     if (PartialMatch(sequenceToMatch))
719                     {
720                         _readAsElements.Add(sequenceToMatch);
721                         _results.Add(sequenceToMatch);
722                     }
723                 }
724             }
725         }
726
727         #endregion
728     }
729 }
```

./Sequences/Sequences.Experiments.cs

```
1  using System;
2  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;
        }

```

```
79          public List<ulong> CreateAllVariants1(params ulong[] sequence)
80          {
81              return Sync.ExecuteWriteOperation(() =>
82              {
83                  if (sequence.IsNullOrEmpty())
84                  {
85                      return new List<ulong>();
86                  }
87                  Links.Unsync.EnsureEachLinkExists(sequence);
88                  if (sequence.Length == 1)
89                  {
90                      return new List<ulong> { sequence[0] };
91                  }
92                  var results = new List<ulong>((int)MathHelpers.Catalan(sequence.Length));
93                  return CreateAllVariants1Core(sequence, results);
94              });
95          }
96
97          private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
98          {
99              if (sequence.Length == 2)
100             {
101                 var link = Links.Unsync.CreateAndUpdate(sequence[0], sequence[1]);
102                 if (link == _constants.Null)
103                 {
104                     throw new NotImplementedException("Creation cancellation is not
                        ↪  implemented.");
105                 }
106                 results.Add(link);
107                 return results;
108             }
109             var innerSequenceLength = sequence.Length - 1;
110             var innerSequence = new ulong[innerSequenceLength];
111             for (var li = 0; li < innerSequenceLength; li++)
112             {
113                 var link = Links.Unsync.CreateAndUpdate(sequence[li], sequence[li + 1]);
114                 if (link == _constants.Null)
115                 {
116                     throw new NotImplementedException("Creation cancellation is not
                        ↪  implemented.");
117                 }
118                 for (var isi = 0; isi < li; isi++)
119                 {
120                     innerSequence[isi] = sequence[isi];
121                 }
122                 innerSequence[li] = link;
123                 for (var isi = li + 1; isi < innerSequenceLength; isi++)
124                 {
125                     innerSequence[isi] = sequence[isi + 1];
126                 }
127                 CreateAllVariants1Core(innerSequence, results);
128             }
129             return results;
130         }
131
132         #endregion
133
134         public HashSet<ulong> Each1(params ulong[] sequence)
135         {
136             var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
137             Each1(link =>
138             {
139                 if (!visitedLinks.Contains(link))
140                 {
141                     visitedLinks.Add(link); // изучить почему случаются повторы
142                 }
143                 return true;
144             }, sequence);
145             return visitedLinks;
146         }
147
148         private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
149         {
150             if (sequence.Length == 2)
151             {
152                 Links.Unsync.Each(sequence[0], sequence[1], handler);
153             }
154             else
```

```
155             {
156                 var innerSequenceLength = sequence.Length - 1;
157                 for (var li = 0; li < innerSequenceLength; li++)
158                 {
159                     var left = sequence[li];
160                     var right = sequence[li + 1];
161                     if (left == 0 && right == 0)
162                     {
163                         continue;
164                     }
165                     var linkIndex = li;
166                     ulong[] innerSequence = null;
167                     Links.Unsync.Each(left, right, doublet =>
168                     {
169                         if (innerSequence == null)
170                         {
171                             innerSequence = new ulong[innerSequenceLength];
172                             for (var isi = 0; isi < linkIndex; isi++)
173                             {
174                                 innerSequence[isi] = sequence[isi];
175                             }
176                             for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
177                             {
178                                 innerSequence[isi] = sequence[isi + 1];
179                             }
180                         }
181                         innerSequence[linkIndex] = doublet;
182                         Each1(handler, innerSequence);
183                         return _constants.Continue;
184                     });
185                 }
186             }
187         }
188 
189         public HashSet<ulong> EachPart(params ulong[] sequence)
190         {
191             var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
192             EachPartCore(link =>
193             {
194                 if (!visitedLinks.Contains(link))
195                 {
196                     visitedLinks.Add(link); // изучить почему случаются повторы
197                 }
198                 return true;
199             }, sequence);
200             return visitedLinks;
201         }
202 
203         public void EachPart(Func<ulong, bool> handler, params ulong[] sequence)
204         {
205             var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
206             EachPartCore(link =>
207             {
208                 if (!visitedLinks.Contains(link))
209                 {
210                     visitedLinks.Add(link); // изучить почему случаются повторы
211                     return handler(link);
212                 }
213                 return true;
214             }, sequence);
215         }
216 
217         private void EachPartCore(Func<ulong, bool> handler, params ulong[] sequence)
218         {
219             if (sequence.IsNullOrEmpty())
220             {
221                 return;
222             }
223             Links.EnsureEachLinkIsAnyOrExists(sequence);
224             if (sequence.Length == 1)
225             {
226                 var link = sequence[0];
227                 if (link > 0)
228                 {
229                     handler(link);
230                 }
231                 else
232                 {
233                     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++)
```

```
682             //             //    AllUsagesCore(sequence[i], results);
683             //
684             //             var filteredResults = new HashSet<ulong>();
685             //             var matcher = new Matcher(this, sequence, filteredResults, null);
686             //             matcher.AddAllPartialMatchedToResults(firstResults);
687             //             return filteredResults;
688             //         }
689             //
690             //         return new HashSet<ulong>();
691             //     });
692             //}
693             
694             public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
695             {
696                 return Sync.ExecuteReadOperation(() =>
697                 {
698                     if (sequence.Length > 0)
699                     {
700                         Links.EnsureEachLinkIsAnyOrExists(sequence);
701                         var firstResults = new HashSet<ulong>();
702                         var lastResults = new HashSet<ulong>();
703                         var first = sequence.First(x => x != _constants.Any);
704                         var last = sequence.Last(x => x != _constants.Any);
705                         AllUsagesCore(first, firstResults);
706                         AllUsagesCore(last, lastResults);
707                         firstResults.IntersectWith(lastResults);
708                         //for (var i = 0; i < sequence.Length; i++)
709                         //    AllUsagesCore(sequence[i], results);
710                         var filteredResults = new HashSet<ulong>();
711                         var matcher = new Matcher(this, sequence, filteredResults, null);
712                         matcher.AddAllPartialMatchedToResults(firstResults);
713                         return filteredResults;
714                     }
715                     return new HashSet<ulong>();
716                 });
717             }
718             
719             public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
    ↪  IList<ulong> sequence)
720             {
721                 return Sync.ExecuteReadOperation(() =>
722                 {
723                     if (sequence.Count > 0)
724                     {
725                         Links.EnsureEachLinkExists(sequence);
726                         var results = new HashSet<LinkIndex>();
727                         //var nextResults = new HashSet<ulong>();
728                         //for (var i = 0; i < sequence.Length; i++)
729                         //{
730                         //    AllUsagesCore(sequence[i], nextResults);
731                         //    if (results.IsNullOrEmpty())
732                         //    {
733                         //        results = nextResults;
734                         //        nextResults = new HashSet<ulong>();
735                         //    }
736                         //    else
737                         //    {
738                         //        results.IntersectWith(nextResults);
739                         //        nextResults.Clear();
740                         //    }
741                         //}
742                         var collector1 = new AllUsagesCollector1(Links.Unsync, results);
743                         collector1.Collect(Links.Unsync.GetLink(sequence[0]));
744                         var next = new HashSet<ulong>();
745                         for (var i = 1; i < sequence.Count; i++)
746                         {
747                             var collector = new AllUsagesCollector1(Links.Unsync, next);
748                             collector.Collect(Links.Unsync.GetLink(sequence[i]));
749                             
750                             results.IntersectWith(next);
751                             next.Clear();
752                         }
753                         var filteredResults = new HashSet<ulong>();
754                         var matcher = new Matcher(this, sequence, filteredResults, null,
    ↪  readAsElements);
755                         matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
    ↪  x)); // OrderBy is a Hack
756                         return filteredResults;
```

```
            }
            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)
        {
```

```
901             return Sync.ExecuteReadOperation(() =>
902             {
903                 var visits = new HashSet<ulong>();
904                 var usages = new HashSet<ulong>();
905                 AllBottomUsagesCore(link, visits, usages);
906                 return usages;
907             });
908         }
909 
910         private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
    ↪  usages)
911         {
912             bool handler(ulong doublet)
913             {
914                 if (visits.Add(doublet))
915                 {
916                     AllBottomUsagesCore(doublet, visits, usages);
917                 }
918                 return true;
919             }
920             if (Links.Unsync.Count(_constants.Any, link) == 0)
921             {
922                 usages.Add(link);
923             }
924             else
925             {
926                 Links.Unsync.Each(link, _constants.Any, handler);
927                 Links.Unsync.Each(_constants.Any, link, handler);
928             }
929         }
930 
931         public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
932         {
933             if (Options.UseSequenceMarker)
934             {
935                 var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
    ↪  Options.MarkedSequenceMatcher, symbol);
936                 return counter.Count();
937             }
938             else
939             {
940                 var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
    ↪  symbol);
941                 return counter.Count();
942             }
943         }
944 
945         private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<ulong, bool>
    ↪  outerHandler)
946         {
947             bool handler(ulong doublet)
948             {
949                 if (usages.Add(doublet))
950                 {
951                     if (!outerHandler(doublet))
952                     {
953                         return false;
954                     }
955                     if (!AllUsagesCore1(doublet, usages, outerHandler))
956                     {
957                         return false;
958                     }
959                 }
960                 return true;
961             }
962             return Links.Unsync.Each(link, _constants.Any, handler)
963                 && Links.Unsync.Each(_constants.Any, link, handler);
964         }
965 
966         public void CalculateAllUsages(ulong[] totals)
967         {
968             var calculator = new AllUsagesCalculator(Links, totals);
969             calculator.Calculate();
970         }
971 
972         public void CalculateAllUsages2(ulong[] totals)
973         {
974             var calculator = new AllUsagesCalculator2(Links, totals);
```

```
975             calculator.Calculate();
976         }
977 
978         private class AllUsagesCalculator
979         {
980             private readonly SynchronizedLinks<ulong> _links;
981             private readonly ulong[] _totals;
982 
983             public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
984             {
985                 _links = links;
986                 _totals = totals;
987             }
988 
989             public void Calculate() => _links.Each(_constants.Any, _constants.Any,
                 ↪ CalculateCore);
990 
991             private bool CalculateCore(ulong link)
992             {
993                 if (_totals[link] == 0)
994                 {
995                     var total = 1UL;
996                     _totals[link] = total;
997                     var visitedChildren = new HashSet<ulong>();
998                     bool linkCalculator(ulong child)
999                     {
1000                        if (link != child && visitedChildren.Add(child))
1001                        {
1002                            total += _totals[child] == 0 ? 1 : _totals[child];
1003                        }
1004                        return true;
1005                    }
1006                    _links.Unsync.Each(link, _constants.Any, linkCalculator);
1007                    _links.Unsync.Each(_constants.Any, link, linkCalculator);
1008                    _totals[link] = total;
1009                }
1010                return true;
1011            }
1012        }
1013 
1014        private class AllUsagesCalculator2
1015        {
1016            private readonly SynchronizedLinks<ulong> _links;
1017            private readonly ulong[] _totals;
1018 
1019            public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1020            {
1021                _links = links;
1022                _totals = totals;
1023            }
1024 
1025            public void Calculate() => _links.Each(_constants.Any, _constants.Any,
                 ↪ CalculateCore);
1026 
1027            private bool IsElement(ulong link)
1028            {
1029                //_linksInSequence.Contains(link) ||
1030                return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
                     ↪ link;
1031            }
1032 
1033            private bool CalculateCore(ulong link)
1034            {
1035                // TODO: Проработать защиту от зацикливания
1036                // Основано на SequenceWalker.WalkLeft
1037                Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1038                Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1039                Func<ulong, bool> isElement = IsElement;
1040                void visitLeaf(ulong parent)
1041                {
1042                    if (link != parent)
1043                    {
1044                        _totals[parent]++;
1045                    }
1046                }
1047                void visitNode(ulong parent)
1048                {
1049                    if (link != parent)
1050                    {
```

```
                        _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;
            }
```

```
1131
1132            public ulong Collect(IList<ulong> link)
1133            {
1134                var linkIndex = _links.GetIndex(link);
1135                if (_usages.Add(linkIndex))
1136                {
1137                    _links.Each(Collect, _constants.Any, linkIndex);
1138                }
1139                return _continue;
1140            }
1141        }
1142
1143        private class AllUsagesCollector2
1144        {
1145            private readonly ILinks<ulong> _links;
1146            private readonly BitString _usages;
1147
1148            public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
1149            {
1150                _links = links;
1151                _usages = usages;
1152            }
1153
1154            public bool Collect(ulong link)
1155            {
1156                if (_usages.Add((long)link))
1157                {
1158                    _links.Each(link, _constants.Any, Collect);
1159                    _links.Each(_constants.Any, link, Collect);
1160                }
1161                return true;
1162            }
1163        }
1164
1165        private class AllUsagesIntersectingCollector
1166        {
1167            private readonly SynchronizedLinks<ulong> _links;
1168            private readonly HashSet<ulong> _intersectWith;
1169            private readonly HashSet<ulong> _usages;
1170            private readonly HashSet<ulong> _enter;
1171
1172            public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
                ↪  intersectWith, HashSet<ulong> usages)
1173            {
1174                _links = links;
1175                _intersectWith = intersectWith;
1176                _usages = usages;
1177                _enter = new HashSet<ulong>(); // защита от зацикливания
1178            }
1179
1180            public bool Collect(ulong link)
1181            {
1182                if (_enter.Add(link))
1183                {
1184                    if (_intersectWith.Contains(link))
1185                    {
1186                        _usages.Add(link);
1187                    }
1188                    _links.Unsync.Each(link, _constants.Any, Collect);
1189                    _links.Unsync.Each(_constants.Any, link, Collect);
1190                }
1191                return true;
1192            }
1193        }
1194
1195        private void CloseInnerConnections(Action<ulong> handler, ulong left, ulong right)
1196        {
1197            TryStepLeftUp(handler, left, right);
1198            TryStepRightUp(handler, right, left);
1199        }
1200
1201        private void AllCloseConnections(Action<ulong> handler, ulong left, ulong right)
1202        {
1203            // Direct
1204            if (left == right)
1205            {
1206                handler(left);
1207            }
1208            var doublet = Links.Unsync.SearchOrDefault(left, right);
1209            if (doublet != _constants.Null)
```

```
1210            {
1211                handler(doublet);
1212            }
1213            // Inner
1214            CloseInnerConnections(handler, left, right);
1215            // Outer
1216            StepLeft(handler, left, right);
1217            StepRight(handler, left, right);
1218            PartialStepRight(handler, left, right);
1219            PartialStepLeft(handler, left, right);
1220        }
1221
1222        private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
        ↪  HashSet<ulong> previousMatchings, long startAt)
1223        {
1224            if (startAt >= sequence.Length) // ?
1225            {
1226                return previousMatchings;
1227            }
1228            var secondLinkUsages = new HashSet<ulong>();
1229            AllUsagesCore(sequence[startAt], secondLinkUsages);
1230            secondLinkUsages.Add(sequence[startAt]);
1231            var matchings = new HashSet<ulong>();
1232            //for (var i = 0; i < previousMatchings.Count; i++)
1233            foreach (var secondLinkUsage in secondLinkUsages)
1234            {
1235                foreach (var previousMatching in previousMatchings)
1236                {
1237                    //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
                    ↪  secondLinkUsage);
1238                    StepRight(matchings.AddAndReturnVoid, previousMatching, secondLinkUsage);
1239                    TryStepRightUp(matchings.AddAndReturnVoid, secondLinkUsage,
                    ↪  previousMatching);
1240                    //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
                    ↪  sequence[startAt]); // почему-то эта ошибочная запись приводит к
                    ↪  желаемым результам.
1241                    PartialStepRight(matchings.AddAndReturnVoid, previousMatching,
                    ↪  secondLinkUsage);
1242                }
1243            }
1244            if (matchings.Count == 0)
1245            {
1246                return matchings;
1247            }
1248            return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
1249        }
1250
1251        private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
        ↪  links, params ulong[] sequence)
1252        {
1253            if (sequence == null)
1254            {
1255                return;
1256            }
1257            for (var i = 0; i < sequence.Length; i++)
1258            {
1259                if (sequence[i] != _constants.Any && sequence[i] != ZeroOrMany &&
                ↪  !links.Exists(sequence[i]))
1260                {
1261                    throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                    ↪  $"patternSequence[{i}]");
1262                }
1263            }
1264        }
1265
1266        // Pattern Matching -> Key To Triggers
1267        public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1268        {
1269            return Sync.ExecuteReadOperation(() =>
1270            {
1271                patternSequence = Simplify(patternSequence);
1272                if (patternSequence.Length > 0)
1273                {
1274                    EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1275                    var uniqueSequenceElements = new HashSet<ulong>();
1276                    for (var i = 0; i < patternSequence.Length; i++)
1277                    {
```

```
                    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
```

```
1661            }
1662
1663            struct PatternBlock
1664            {
1665                public PatternBlockType Type;
1666                public long Start;
1667                public long Stop;
1668            }
1669
1670            private readonly List<PatternBlock> _pattern;
1671            private int _patternPosition;
1672            private long _sequencePosition;
1673
1674            #endregion
1675
1676            public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
                ↪  HashSet<LinkIndex> results)
1677                : base(sequences.Links.Unsync)
1678            {
1679                _sequences = sequences;
1680                _patternSequence = patternSequence;
1681                _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                    ↪  _constants.Any && x != ZeroOrMany));
1682                _results = results;
1683                _pattern = CreateDetailedPattern();
1684            }
1685
1686            protected override bool IsElement(IList<ulong> link) =>
                ↪  _linksInSequence.Contains(Links.GetIndex(link)) || base.IsElement(link);
1687
1688            public bool PatternMatch(LinkIndex sequenceToMatch)
1689            {
1690                _patternPosition = 0;
1691                _sequencePosition = 0;
1692                foreach (var part in Walk(sequenceToMatch))
1693                {
1694                    if (!PatternMatchCore(Links.GetIndex(part)))
1695                    {
1696                        break;
1697                    }
1698                }
1699                return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
                    ↪  - 1 && _pattern[_patternPosition].Start == 0);
1700            }
1701
1702            private List<PatternBlock> CreateDetailedPattern()
1703            {
1704                var pattern = new List<PatternBlock>();
1705                var patternBlock = new PatternBlock();
1706                for (var i = 0; i < _patternSequence.Length; i++)
1707                {
1708                    if (patternBlock.Type == PatternBlockType.Undefined)
1709                    {
1710                        if (_patternSequence[i] == _constants.Any)
1711                        {
1712                            patternBlock.Type = PatternBlockType.Gap;
1713                            patternBlock.Start = 1;
1714                            patternBlock.Stop = 1;
1715                        }
1716                        else if (_patternSequence[i] == ZeroOrMany)
1717                        {
1718                            patternBlock.Type = PatternBlockType.Gap;
1719                            patternBlock.Start = 0;
1720                            patternBlock.Stop = long.MaxValue;
1721                        }
1722                        else
1723                        {
1724                            patternBlock.Type = PatternBlockType.Elements;
1725                            patternBlock.Start = i;
1726                            patternBlock.Stop = i;
1727                        }
1728                    }
1729                    else if (patternBlock.Type == PatternBlockType.Elements)
1730                    {
1731                        if (_patternSequence[i] == _constants.Any)
1732                        {
1733                            pattern.Add(patternBlock);
1734                            patternBlock = new PatternBlock
1735                            {
1736                                Type = PatternBlockType.Gap,
```

```
                        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

```
//#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

```
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

```
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

```
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; }
```

```
23          public bool EnforceSingleSequenceVersionOnWriteBasedOnNew { get; set; }
24
25          public MarkedSequenceCreteriaMatcher<TLink> MarkedSequenceMatcher { get; set; }
26          public IConverter<IList<TLink>, TLink> LinksToSequenceConverter { get; set; }
27          public SequencesIndexer<TLink> Indexer { get; set; }
28
29          // TODO: Реализовать компактификацию при чтении
30          //public bool EnforceSingleSequenceVersionOnRead { get; set; }
31          //public bool UseRequestMarker { get; set; }
32          //public bool StoreRequestResults { get; set; }
33
34          public void InitOptions(ISynchronizedLinks<TLink> links)
35          {
36              if (UseSequenceMarker)
37              {
38                  if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
39                  {
40                      SequenceMarkerLink = links.CreatePoint();
41                  }
42                  else
43                  {
44                      if (!links.Exists(SequenceMarkerLink))
45                      {
46                          var link = links.CreatePoint();
47                          if (!_equalityComparer.Equals(link, SequenceMarkerLink))
48                          {
49                              throw new InvalidOperationException("Cannot recreate sequence marker
                                ↪  link.");
50                          }
51                      }
52                  }
53                  if (MarkedSequenceMatcher == null)
54                  {
55                      MarkedSequenceMatcher = new MarkedSequenceCreteriaMatcher<TLink>(links,
                        ↪  SequenceMarkerLink);
56                  }
57              }
58              var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
59              if (UseCompression)
60              {
61                  if (LinksToSequenceConverter == null)
62                  {
63                      ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
64                      if (UseSequenceMarker)
65                      {
66                          totalSequenceSymbolFrequencyCounter = new
                            ↪  TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
                            ↪  MarkedSequenceMatcher);
67                      }
68                      else
69                      {
70                          totalSequenceSymbolFrequencyCounter = new
                            ↪  TotalSequenceSymbolFrequencyCounter<TLink>(links);
71                      }
72                      var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
                        ↪  totalSequenceSymbolFrequencyCounter);
73                      var compressingConverter = new CompressingConverter<TLink>(links,
                        ↪  balancedVariantConverter, doubletFrequenciesCache);
74                      LinksToSequenceConverter = compressingConverter;
75                  }
76              }
77              else
78              {
79                  if (LinksToSequenceConverter == null)
80                  {
81                      LinksToSequenceConverter = balancedVariantConverter;
82                  }
83              }
84              if (UseIndex && Indexer == null)
85              {
86                  Indexer = new SequencesIndexer<TLink>(links);
87              }
88          }
89
90          public void ValidateOptions()
91          {
92              if (UseGarbageCollection && !UseSequenceMarker)
93              {
```

```
                throw new NotSupportedException("To use garbage collection UseSequenceMarker
                ↪  option must be on.");
            }
        }
    }
}
```

./Sequences/UnicodeMap.cs

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

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

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

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

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

        public void Init()
        {
            if (_initialized)
            {
                return;
            }
            _initialized = true;
            var firstLink = _links.CreatePoint();
            if (firstLink != FirstCharLink)
            {
                _links.Delete(firstLink);
            }
            else
            {
                for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
                {
                    // From NIL to It (NIL -> Character) transformation meaning, (or infinite
                    ↪  amount of NIL characters before actual Character)
                    var createdLink = _links.CreatePoint();
                    _links.Update(createdLink, firstLink, createdLink);
                    if (createdLink != i)
                    {
                        throw new InvalidOperationException("Unable to initialize UTF 16
                        ↪  table.");
                    }
                }
            }
        }

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

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

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

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

        public static string FromLinksToString(IList<ulong> linksList)
        {
```

```
71              var sb = new StringBuilder();
72              for (int i = 0; i < linksList.Count; i++)
73              {
74                  sb.Append(FromLinkToChar(linksList[i]));
75              }
76              return sb.ToString();
77          }
78
79          public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
80          {
81              var sb = new StringBuilder();
82              if (links.Exists(link))
83              {
84                  StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
85                      x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
                        ↪  element =>
86                      {
87                          sb.Append(FromLinkToChar(element));
88                          return true;
89                      });
90              }
91              return sb.ToString();
92          }
93
94          public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
            ↪  chars.Length);
95
96          public static ulong[] FromCharsToLinkArray(char[] chars, int count)
97          {
98              // char array to ulong array
99              var linksSequence = new ulong[count];
100             for (var i = 0; i < count; i++)
101             {
102                 linksSequence[i] = FromCharToLink(chars[i]);
103             }
104             return linksSequence;
105         }
106
107         public static ulong[] FromStringToLinkArray(string sequence)
108         {
109             // char array to ulong array
110             var linksSequence = new ulong[sequence.Length];
111             for (var i = 0; i < sequence.Length; i++)
112             {
113                 linksSequence[i] = FromCharToLink(sequence[i]);
114             }
115             return linksSequence;
116         }
117
118         public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
119         {
120             var result = new List<ulong[]>();
121             var offset = 0;
122             while (offset < sequence.Length)
123             {
124                 var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
125                 var relativeLength = 1;
126                 var absoluteLength = offset + relativeLength;
127                 while (absoluteLength < sequence.Length &&
128                        currentCategory ==
                           ↪  CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
129                 {
130                     relativeLength++;
131                     absoluteLength++;
132                 }
133                 // char array to ulong array
134                 var innerSequence = new ulong[relativeLength];
135                 var maxLength = offset + relativeLength;
136                 for (var i = offset; i < maxLength; i++)
137                 {
138                     innerSequence[i - offset] = FromCharToLink(sequence[i]);
139                 }
140                 result.Add(innerSequence);
141                 offset += relativeLength;
142             }
143             return result;
144         }
145
146         public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
```

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

./Sequences/Walkers/LeftSequenceWalker.cs

```
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

```
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

```
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

```
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

```
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;
        }

```

```
12          public static void DeleteStack<TLink>(this ILinks<TLink> links, TLink stack) =>
            ↪ links.Delete(stack);
13      }
14  }
```

./SynchronizedLinks.cs

```
1   using System;
2   using System.Collections.Generic;
3   using Platform.Data.Constants;
4   using Platform.Data.Doublets;
5   using Platform.Threading.Synchronization;
6
7   namespace Platform.Data.Doublets
8   {
9       /// <remarks>
10      /// TODO: Autogeneration of synchronized wrapper (decorator).
11      /// TODO: Try to unfold code of each method using IL generation for performance improvements.
12      /// TODO: Or even to unfold multiple layers of implementations.
13      /// </remarks>
14      public class SynchronizedLinks<T> : ISynchronizedLinks<T>
15      {
16          public LinksCombinedConstants<T, T, int> Constants { get; }
17          public ISynchronization SyncRoot { get; }
18          public ILinks<T> Sync { get; }
19          public ILinks<T> Unsync { get; }
20
21          public SynchronizedLinks(ILinks<T> links) : this(new ReaderWriterLockSynchronization(),
            ↪ links) { }
22
23          public SynchronizedLinks(ISynchronization synchronization, ILinks<T> links)
24          {
25              SyncRoot = synchronization;
26              Sync = this;
27              Unsync = links;
28              Constants = links.Constants;
29          }
30
31          public T Count(IList<T> restriction) => SyncRoot.ExecuteReadOperation(restriction,
            ↪ Unsync.Count);
32          public T Each(Func<IList<T>, T> handler, IList<T> restrictions) =>
            ↪ SyncRoot.ExecuteReadOperation(handler, restrictions, (handler1, restrictions1) =>
            ↪ Unsync.Each(handler1, restrictions1));
33          public T Create() => SyncRoot.ExecuteWriteOperation(Unsync.Create);
34          public T Update(IList<T> restrictions) => SyncRoot.ExecuteWriteOperation(restrictions,
            ↪ Unsync.Update);
35          public void Delete(T link) => SyncRoot.ExecuteWriteOperation(link, Unsync.Delete);
36
37          //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
            ↪ IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
38          //{
39          //    if (restriction != null && substitution != null &&
            ↪ !substitution.EqualTo(restriction))
40          //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
            ↪ substitution, substitutedHandler, Unsync.Trigger);
41
42          //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
            ↪ substitutedHandler, Unsync.Trigger);
43          //}
44      }
45  }
```

./UInt64Link.cs

```
1   using System;
2   using System.Collections;
3   using System.Collections.Generic;
4   using Platform.Exceptions;
5   using Platform.Ranges;
6   using Platform.Helpers.Singletons;
7   using Platform.Data.Constants;
8
9   namespace Platform.Data.Doublets
10  {
11      /// <summary>
12      /// Структура описывающая уникальную связь.
13      /// </summary>
14      public struct UInt64Link : IEquatable<UInt64Link>, IReadOnlyList<ulong>, IList<ulong>
15      {
16          private static readonly LinksCombinedConstants<bool, ulong, int> _constants =
            ↪ Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
```

```
17
18          private const int Length = 3;
19
20          public readonly ulong Index;
21          public readonly ulong Source;
22          public readonly ulong Target;
23
24          public static readonly UInt64Link Null = new UInt64Link();
25
26          public UInt64Link(params ulong[] values)
27          {
28              Index = values.Length > _constants.IndexPart ? values[_constants.IndexPart] :
                ↪  _constants.Null;
29              Source = values.Length > _constants.SourcePart ? values[_constants.SourcePart] :
                ↪  _constants.Null;
30              Target = values.Length > _constants.TargetPart ? values[_constants.TargetPart] :
                ↪  _constants.Null;
31          }
32
33          public UInt64Link(IList<ulong> values)
34          {
35              Index = values.Count > _constants.IndexPart ? values[_constants.IndexPart] :
                ↪  _constants.Null;
36              Source = values.Count > _constants.SourcePart ? values[_constants.SourcePart] :
                ↪  _constants.Null;
37              Target = values.Count > _constants.TargetPart ? values[_constants.TargetPart] :
                ↪  _constants.Null;
38          }
39
40          public UInt64Link(ulong index, ulong source, ulong target)
41          {
42              Index = index;
43              Source = source;
44              Target = target;
45          }
46
47          public UInt64Link(ulong source, ulong target)
48              : this(_constants.Null, source, target)
49          {
50              Source = source;
51              Target = target;
52          }
53
54          public static UInt64Link Create(ulong source, ulong target) => new UInt64Link(source,
            ↪  target);
55
56          public override int GetHashCode() => (Index, Source, Target).GetHashCode();
57
58          public bool IsNull() => Index == _constants.Null
59                                  && Source == _constants.Null
60                                  && Target == _constants.Null;
61
62          public override bool Equals(object other) => other is UInt64Link &&
            ↪  Equals((UInt64Link)other);
63
64          public bool Equals(UInt64Link other) => Index == other.Index
65                                                  && Source == other.Source
66                                                  && Target == other.Target;
67
68          public static string ToString(ulong index, ulong source, ulong target) => $"({index}:
            ↪  {source}->{target})";
69
70          public static string ToString(ulong source, ulong target) => $"({source}->{target})";
71
72          public static implicit operator ulong[](UInt64Link link) => link.ToArray();
73
74          public static implicit operator UInt64Link(ulong[] linkArray) => new
            ↪  UInt64Link(linkArray);
75
76          public ulong[] ToArray()
77          {
78              var array = new ulong[Length];
79              CopyTo(array, 0);
80              return array;
81          }
82
83          public override string ToString() => Index == _constants.Null ? ToString(Source, Target)
            ↪  : ToString(Index, Source, Target);
84
85          #region IList
```

```
86
87          public ulong this[int index]
88          {
89              get
90              {
91                  Ensure.Always.ArgumentInRange(index, new Range<int>(0, Length - 1),
                    ↪  nameof(index));
92                  if (index == _constants.IndexPart)
93                  {
94                      return Index;
95                  }
96                  if (index == _constants.SourcePart)
97                  {
98                      return Source;
99                  }
100                 if (index == _constants.TargetPart)
101                 {
102                     return Target;
103                 }
104                 throw new NotSupportedException(); // Impossible path due to
                    ↪  Ensure.ArgumentInRange
105             }
106             set => throw new NotSupportedException();
107         }
108
109         public int Count => Length;
110
111         public bool IsReadOnly => true;
112
113         IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
114
115         public IEnumerator<ulong> GetEnumerator()
116         {
117             yield return Index;
118             yield return Source;
119             yield return Target;
120         }
121
122         public void Add(ulong item) => throw new NotSupportedException();
123
124         public void Clear() => throw new NotSupportedException();
125
126         public bool Contains(ulong item) => IndexOf(item) >= 0;
127
128         public void CopyTo(ulong[] array, int arrayIndex)
129         {
130             Ensure.Always.ArgumentNotNull(array, nameof(array));
131             Ensure.Always.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
                ↪  nameof(arrayIndex));
132             if (arrayIndex + Length > array.Length)
133             {
134                 throw new ArgumentException();
135             }
136             array[arrayIndex++] = Index;
137             array[arrayIndex++] = Source;
138             array[arrayIndex] = Target;
139         }
140
141         public bool Remove(ulong item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
142
143         public int IndexOf(ulong item)
144         {
145             if (Index == item)
146             {
147                 return _constants.IndexPart;
148             }
149             if (Source == item)
150             {
151                 return _constants.SourcePart;
152             }
153             if (Target == item)
154             {
155                 return _constants.TargetPart;
156             }
157
158             return -1;
159         }
160
161         public void Insert(int index, ulong item) => throw new NotSupportedException();
162
```

```
163         public void RemoveAt(int index) => throw new NotSupportedException();
164
165         #endregion
166     }
167 }
```

./UInt64LinkExtensions.cs

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

./UInt64LinksExtensions.cs

```
1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using Platform.Helpers.Singletons;
5  using Platform.Data.Constants;
6  using Platform.Data.Exceptions;
7  using Platform.Data.Doublets.Sequences;
8
9  namespace Platform.Data.Doublets
10 {
11     public static class UInt64LinksExtensions
12     {
13         public static readonly LinksCombinedConstants<bool, ulong, int> Constants =
           ↪  Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
14
15         public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
16
17         public static void EnsureEachLinkExists(this ILinks<ulong> links, IList<ulong> sequence)
18         {
19             if (sequence == null)
20             {
21                 return;
22             }
23             for (var i = 0; i < sequence.Count; i++)
24             {
25                 if (!links.Exists(sequence[i]))
26                 {
27                     throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                       ↪  $"sequence[{i}]");
28                 }
29             }
30         }
31
32         public static void EnsureEachLinkIsAnyOrExists(this ILinks<ulong> links, IList<ulong>
           ↪  sequence)
33         {
34             if (sequence == null)
35             {
36                 return;
37             }
38             for (var i = 0; i < sequence.Count; i++)
39             {
40                 if (sequence[i] != Constants.Any && !links.Exists(sequence[i]))
41                 {
42                     throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                       ↪  $"sequence[{i}]");
43                 }
44             }
45         }
46
47         public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
48         {
49             if (sequence == null)
50             {
51                 return false;
52             }
53             var constants = links.Constants;
54             for (var i = 0; i < sequence.Length; i++)
55             {
56                 if (sequence[i] == constants.Any)
57                 {
```

```
                    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

```
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
```

```
47          ///     {
48          ///         get
49          ///         {
50          ///             return (ulong) mask & TransactionIdCombined;
51          ///         }
52          ///     }
53          ///
54          ///     public UniqueTimestamp Timestamp
55          ///     {
56          ///         get
57          ///         {
58          ///             return (UniqueTimestamp)mask & TransactionIdCombined;
59          ///         }
60          ///     }
61          ///
62          ///     public TransactionItemType Type
63          ///     {
64          ///         get
65          ///         {
66          ///             // Использовать по одному биту из TransactionId и Timestamp,
67          ///             // для значения в 2 бита, которое представляет тип операции
68          ///             throw new NotImplementedException();
69          ///         }
70          ///     }
71          /// }
72          ///
73          /// private struct Transition
74          /// {
75          ///     public TransitionHeader Header;
76          ///     public Link Source;
77          ///     public Link Linker;
78          ///     public Link Target;
79          /// }
80          ///
81          /// </remarks>
82          public struct Transition
83          {
84              public static readonly long Size = StructureHelpers.SizeOf<Transition>();
85
86              public readonly ulong TransactionId;
87              public readonly UInt64Link Before;
88              public readonly UInt64Link After;
89              public readonly Timestamp Timestamp;
90
91              public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
                ↪  transactionId, UInt64Link before, UInt64Link after)
92              {
93                  TransactionId = transactionId;
94                  Before = before;
95                  After = after;
96                  Timestamp = uniqueTimestampFactory.Create();
97              }
98
99              public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
                ↪  transactionId, UInt64Link before)
100                 : this(uniqueTimestampFactory, transactionId, before, default)
101             {
102             }
103
104             public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId)
105                 : this(uniqueTimestampFactory, transactionId, default, default)
106             {
107             }
108
109             public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
                ↪  {After}";
110         }
111
112         /// <remarks>
113         /// Другие варианты реализации транзакций (атомарности):
114         ///     1. Разделение хранения значения связи ((Source Target) или (Source Linker
            ↪  Target)) и индексов.
115         ///     2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
            ↪  потребуется решить вопрос
116         ///        со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
            ↪  пересечениями идентификаторов.
117         ///
118         /// Где хранить промежуточный список транзакций?
```

```
119         ///
120         /// В оперативной памяти:
121         ///  Минусы:
122         ///     1. Может усложнить систему, если она будет функционировать самостоятельно,
123         ///     так как нужно отдельно выделять память под список трансформаций.
124         ///     2. Выделенной оперативной памяти может не хватить, в том случае,
125         ///     если транзакция использует слишком много трансформаций.
126         ///         -> Можно использовать жёсткий диск для слишком длинных транзакций.
127         ///         -> Максимальный размер списка трансформаций можно ограничить / задать
            ↪ константой.
128         ///     3. При подтверждении транзакции (Commit) все трансформации записываются разом
            ↪ создавая задержку.
129         ///
130         /// На жёстком диске:
131         ///  Минусы:
132         ///     1. Длительный отклик, на запись каждой трансформации.
133         ///     2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
134         ///         -> Это может решаться упаковкой/исключением дублирующих операций.
135         ///         -> Также это может решаться тем, что короткие транзакции вообще
136         ///            не будут записываться в случае отката.
137         ///     3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
            ↪ операции (трансформации)
138         ///        будут записаны в лог.
139         ///
140         /// </remarks>
141         public class Transaction : DisposableBase
142         {
143             private readonly Queue<Transition> _transitions;
144             private readonly UInt64LinksTransactionsLayer _layer;
145             public bool IsCommitted { get; private set; }
146             public bool IsReverted { get; private set; }
147 
148             public Transaction(UInt64LinksTransactionsLayer layer)
149             {
150                 _layer = layer;
151                 if (_layer._currentTransactionId != 0)
152                 {
153                     throw new NotSupportedException("Nested transactions not supported.");
154                 }
155                 IsCommitted = false;
156                 IsReverted = false;
157                 _transitions = new Queue<Transition>();
158                 SetCurrentTransaction(layer, this);
159             }
160 
161             public void Commit()
162             {
163                 EnsureTransactionAllowsWriteOperations(this);
164                 while (_transitions.Count > 0)
165                 {
166                     var transition = _transitions.Dequeue();
167                     _layer._transitions.Enqueue(transition);
168                 }
169                 _layer._lastCommitedTransactionId = _layer._currentTransactionId;
170                 IsCommitted = true;
171             }
172 
173             private void Revert()
174             {
175                 EnsureTransactionAllowsWriteOperations(this);
176                 var transitionsToRevert = new Transition[_transitions.Count];
177                 _transitions.CopyTo(transitionsToRevert, 0);
178                 for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
179                 {
180                     _layer.RevertTransition(transitionsToRevert[i]);
181                 }
182                 IsReverted = true;
183             }
184 
185             public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
                ↪ Transaction transaction)
186             {
187                 layer._currentTransactionId = layer._lastCommitedTransactionId + 1;
188                 layer._currentTransactionTransitions = transaction._transitions;
189                 layer._currentTransaction = transaction;
190             }
191 
192             public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
193             {
```

```
194                 if (transaction.IsReverted)
195                 {
196                     throw new InvalidOperationException("Transation is reverted.");
197                 }
198                 if (transaction.IsCommitted)
199                 {
200                     throw new InvalidOperationException("Transation is commited.");
201                 }
202             }
203 
204             protected override void DisposeCore(bool manual, bool wasDisposed)
205             {
206                 if (!wasDisposed && _layer != null && !_layer.IsDisposed)
207                 {
208                     if (!IsCommitted && !IsReverted)
209                     {
210                         Revert();
211                     }
212                     _layer.ResetCurrentTransation();
213                 }
214             }
215 
216             // TODO: THIS IS EXCEPTION WORKAROUND, REMOVE IT THEN
                ↪ https://github.com/linksplatform/Disposables/issues/13 FIXED
217             protected override bool AllowMultipleDisposeCalls => true;
218         }
219 
220         public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
221 
222         private readonly string _logAddress;
223         private readonly FileStream _log;
224         private readonly Queue<Transition> _transitions;
225         private readonly UniqueTimestampFactory _uniqueTimestampFactory;
226         private Task _transitionsPusher;
227         private Transition _lastCommitedTransition;
228         private ulong _currentTransactionId;
229         private Queue<Transition> _currentTransactionTransitions;
230         private Transaction _currentTransaction;
231         private ulong _lastCommitedTransactionId;
232 
233         public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
234             : base(links)
235         {
236             if (string.IsNullOrWhiteSpace(logAddress))
237             {
238                 throw new ArgumentNullException(nameof(logAddress));
239             }
240             // В первой строке файла хранится последняя закоммиченную транзакцию.
241             // При запуске это используется для проверки удачного закрытия файла лога.
242             // In the first line of the file the last committed transaction is stored.
243             // On startup, this is used to check that the log file is successfully closed.
244             var lastCommitedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
245             var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
246             if (!lastCommitedTransition.Equals(lastWrittenTransition))
247             {
248                 Dispose();
249                 throw new NotSupportedException("Database is damaged, autorecovery is not
                    ↪ supported yet.");
250             }
251             if (lastCommitedTransition.Equals(default(Transition)))
252             {
253                 FileHelpers.WriteFirst(logAddress, lastCommitedTransition);
254             }
255             _lastCommitedTransition = lastCommitedTransition;
256             // TODO: Think about a better way to calculate or store this value
257             var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
258             _lastCommitedTransactionId = allTransitions.Max(x => x.TransactionId);
259             _uniqueTimestampFactory = new UniqueTimestampFactory();
260             _logAddress = logAddress;
261             _log = FileHelpers.Append(logAddress);
262             _transitions = new Queue<Transition>();
263             _transitionsPusher = new Task(TransitionsPusher);
264             _transitionsPusher.Start();
265         }
266 
267         public IList<ulong> GetLinkValue(ulong link) => Links.GetLink(link);
268 
269         public override ulong Create()
270         {
```

```
271             var createdLinkIndex = Links.Create();
272             var createdLink = new UInt64Link(Links.GetLink(createdLinkIndex));
273             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                ↪  default, createdLink));
274             return createdLinkIndex;
275         }
276 
277         public override ulong Update(IList<ulong> parts)
278         {
279             var beforeLink = new UInt64Link(Links.GetLink(parts[Constants.IndexPart]));
280             parts[Constants.IndexPart] = Links.Update(parts);
281             var afterLink = new UInt64Link(Links.GetLink(parts[Constants.IndexPart]));
282             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                ↪  beforeLink, afterLink));
283             return parts[Constants.IndexPart];
284         }
285 
286         public override void Delete(ulong link)
287         {
288             var deletedLink = new UInt64Link(Links.GetLink(link));
289             Links.Delete(link);
290             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                ↪  deletedLink, default));
291         }
292 
293         [MethodImpl(MethodImplOptions.AggressiveInlining)]
294         private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
            ↪  _transitions;
295 
296         private void CommitTransition(Transition transition)
297         {
298             if (_currentTransaction != null)
299             {
300                 Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
301             }
302             var transitions = GetCurrentTransitions();
303             transitions.Enqueue(transition);
304         }
305 
306         private void RevertTransition(Transition transition)
307         {
308             if (transition.After.IsNull()) // Revert Deletion with Creation
309             {
310                 Links.Create();
311             }
312             else if (transition.Before.IsNull()) // Revert Creation with Deletion
313             {
314                 Links.Delete(transition.After.Index);
315             }
316             else // Revert Update
317             {
318                 Links.Update(new[] { transition.After.Index, transition.Before.Source,
                    ↪  transition.Before.Target });
319             }
320         }
321 
322         private void ResetCurrentTransation()
323         {
324             _currentTransactionId = 0;
325             _currentTransactionTransitions = null;
326             _currentTransaction = null;
327         }
328 
329         private void PushTransitions()
330         {
331             if (_log == null || _transitions == null)
332             {
333                 return;
334             }
335             for (var i = 0; i < _transitions.Count; i++)
336             {
337                 var transition = _transitions.Dequeue();
338 
339                 _log.Write(transition);
340                 _lastCommitedTransition = transition;
341             }
342         }
343 
```

```
344         private void TransitionsPusher()
345         {
346             while (!IsDisposed && _transitionsPusher != null)
347             {
348                 Thread.Sleep(DefaultPushDelay);
349                 PushTransitions();
350             }
351         }
352 
353         public Transaction BeginTransaction() => new Transaction(this);
354 
355         private void DisposeTransitions()
356         {
357             try
358             {
359                 var pusher = _transitionsPusher;
360                 if (pusher != null)
361                 {
362                     _transitionsPusher = null;
363                     pusher.Wait();
364                 }
365                 if (_transitions != null)
366                 {
367                     PushTransitions();
368                 }
369                 Disposable.TryDispose(_log);
370                 FileHelpers.WriteFirst(_logAddress, _lastCommitedTransition);
371             }
372             catch
373             {
374             }
375         }
376 
377         #region DisposalBase
378 
379         protected override void DisposeCore(bool manual, bool wasDisposed)
380         {
381             if (!wasDisposed)
382             {
383                 DisposeTransitions();
384             }
385             base.DisposeCore(manual, wasDisposed);
386         }
387 
388         #endregion
389     }
390 }
```

Index