./Platform.Data.Doublets/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 < Type<TLink>.BitsLength; i++)
            {
                if (_equalityComparer.Equals(Arithmetic.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 Bit.ShiftRight(number, 1);
                if (_equalityComparer.Equals(number, default))
                {
                    break;
                }
            }
            return target;
        }
    }
}
```

./Platform.Data.Doublets/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 IPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
        private readonly IConverter<TLink> _unaryNumberToAddressConverter;

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

        public TLink Convert(Doublet<TLink> doublet)
        {
            var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
            if (_equalityComparer.Equals(link, Links.Constants.Null))
            {
                throw new ArgumentException($"Link 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;
        }
    }
}
```

./Platform.Data.Doublets/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;
        }
    }
}
```

./Platform.Data.Doublets/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 = Arithmetic.Add(result, _unaryToUInt64[source]);
                    target = Links.GetTarget(target);
                }
                result = Arithmetic.Add(result, lastValue);
                return result;
            }
        }
    }
}
```

./Platform.Data.Doublets/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 < Type<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); //
                ↪ Math.Or(target, Math.ShiftLeft(One, powerOf2Index))
            }
            return target;
        }
    }
}
```

./Platform.Data.Doublets/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);
        }
    }
}
```

./Platform.Data.Doublets/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);
        }
    }
}
```

./Platform.Data.Doublets/Decorators/LinksDecoratorBase.cs

```
using System;
using System.Collections.Generic;
using Platform.Data.Constants;

namespace Platform.Data.Doublets.Decorators
{
    public abstract class LinksDecoratorBase<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);
    }
}
```

./Platform.Data.Doublets/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);
        }
    }
}
```

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

```
using System;
using System.Collections.Generic;
using Platform.Disposables;
using Platform.Data.Constants;

namespace Platform.Data.Doublets.Decorators
{
    public abstract class LinksDisposableDecoratorBase<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 Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                Links.DisposeIfPossible();
            }
        }
    }
}
```

./Platform.Data.Doublets/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);
        }
    }
}
```

./Platform.Data.Doublets/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);
        }
    }
}
```

./Platform.Data.Doublets/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);
        }
    }
}
```

./Platform.Data.Doublets/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);
        }
    }
}
```

./Platform.Data.Doublets/Decorators/LinksUniquenessResolver.cs

```
using System.Collections.Generic;

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

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

        public override TLink Update(IList<TLink> restrictions)
        {
            var newLinkAddress = Links.SearchOrDefault(restrictions[Constants.SourcePart],
             ↪ restrictions[Constants.TargetPart]);
            if (_equalityComparer.Equals(newLinkAddress, default))
            {
                return 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;
        }
    }
}
```

./Platform.Data.Doublets/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);
```

```
        }
    }
}
```

./Platform.Data.Doublets/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);
        }
    }
}
```

./Platform.Data.Doublets/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)
            {
                throw new NotSupportedException();
            }
            var updatedLink = restrictions[Constants.IndexPart];
            this.EnsureLinkExists(updatedLink, nameof(Constants.IndexPart));
            var newSource = restrictions[Constants.SourcePart];
            this.EnsureLinkIsItselfOrExists(newSource, nameof(Constants.SourcePart));
            var newTarget = restrictions[Constants.TargetPart];
            this.EnsureLinkIsItselfOrExists(newTarget, nameof(Constants.TargetPart));
            var existedLink = Constants.Null;
            if (newSource != Constants.Itself && newTarget != Constants.Itself)
            {
                existedLink = this.SearchOrDefault(newSource, newTarget);
            }
            if (existedLink == Constants.Null)
            {
                var before = Links.GetLink(updatedLink);
                if (before[Constants.SourcePart] != newSource || before[Constants.TargetPart] != newTarget)
                {
                    Links.Update(updatedLink, newSource == Constants.Itself ? updatedLink : newSource,
                                              newTarget == Constants.Itself ? updatedLink : newTarget);
                }
                return updatedLink;
            }
            else
            {
                // Replace one link with another (replaced link is deleted, children are updated or deleted), it is actually merge operation
                return this.Merge(updatedLink, existedLink);
            }
        }

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

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

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

```
using System.Collections.ObjectModel;

namespace Platform.Data.Doublets.Decorators
{
    /// <remarks>
    /// What does empty pattern (for condition or substitution) mean? Nothing or Everything?
    /// Now we go with nothing. And nothing is something one, but empty, and cannot be changed by itself. But can cause creation (update from nothing) or deletion (update to nothing).
    ///
    /// TODO: Decide to change to IDoubletLinks or not to change. (Better to create DefaultUniLinksBase, that contains logic itself and can be implemented using both IDoubletLinks and ILinks.)
    /// </remarks>
    internal class UniLinks<TLink> : LinksDecoratorBase<TLink>, IUniLinks<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer = EqualityComparer<TLink>.Default;

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

        private struct Transition
        {
            public IList<TLink> Before;
            public IList<TLink> After;

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

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

        // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution (Links-Expression)
        public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>, TLink> matchedHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink> substitutedHandler)
        {
            ////List<Transition> transitions = null;
            ////if (!restriction.IsNullOrEmpty())
            ////{
            ////    // Есть причина делать проход (чтение)
            ////    if (matchedHandler != null)
            ////    {
            ////        if (!substitution.IsNullOrEmpty())
            ////        {
            ////            // restriction => { 0, 0, 0 } | { 0 } // Create
            ////            // substitution => { itself, 0, 0 } | { itself, itself, itself } // Create / Update
            ////            // substitution => { 0, 0, 0 } | { 0 } // Delete
            ////            transitions = new List<Transition>();
            ////            if (Equals(substitution[Constants.IndexPart], Constants.Null))
            ////            {
            ////                // If index is Null, that means we always ignore every other value (they are also Null by definition)
            ////                var matchDecision = matchedHandler(, NullLink);
            ////                if (Equals(matchDecision, Constants.Break))
            ////                    return false;
            ////                if (!Equals(matchDecision, Constants.Skip))
            ////                    transitions.Add(new Transition(matchedLink, newValue));
            ////            }
            ////            else
            ////            {
            ////                Func<T, bool> handler;
            ////                handler = link =>
            ////                {
            ////                    var matchedLink = Memory.GetLinkValue(link);
            ////                    var newValue = Memory.GetLinkValue(link);
            ////                    newValue[Constants.IndexPart] = Constants.Itself;
            ////                    newValue[Constants.SourcePart] = Equals(substitution[Constants.SourcePart], Constants.Itself) ? matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
```

```
////                    newValue[Constants.TargetPart] =
↪  Equals(substitution[Constants.TargetPart], Constants.Itself) ?
↪  matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
////                    var matchDecision = matchedHandler(matchedLink, newValue);
////                    if (Equals(matchDecision, Constants.Break))
////                        return false;
////                    if (!Equals(matchDecision, Constants.Skip))
////                        transitions.Add(new Transition(matchedLink, newValue));
////                    return true;
////                };
////                if (!Memory.Each(handler, restriction))
////                    return Constants.Break;
////            }
////        }
////        else
////        {
////            Func<T, bool> handler = link =>
////            {
////                var matchedLink = Memory.GetLinkValue(link);
////                var matchDecision = matchedHandler(matchedLink, matchedLink);
////                return !Equals(matchDecision, Constants.Break);
////            };
////            if (!Memory.Each(handler, restriction))
////                return Constants.Break;
////        }
////    }
////    else
////    {
////        if (substitution != null)
////        {
////            transitions = new List<IList<T>>();
////            Func<T, bool> handler = link =>
////            {
////                var matchedLink = Memory.GetLinkValue(link);
////                transitions.Add(matchedLink);
////                return true;
////            };
////            if (!Memory.Each(handler, restriction))
////                return Constants.Break;
////        }
////        else
////        {
////            return Constants.Continue;
////        }
////    }
////}
////if (substitution != null)
////{
////    // Есть причина делать замену (запись)
////    if (substitutedHandler != null)
////    {
////    }
////    else
////    {
////    }
////}
////return Constants.Continue;

//if (restriction.IsNullOrEmpty()) // Create
//{
//    substitution[Constants.IndexPart] = Memory.AllocateLink();
//    Memory.SetLinkValue(substitution);
//}
//else if (substitution.IsNullOrEmpty()) // Delete
//{
//    Memory.FreeLink(restriction[Constants.IndexPart]);
//}
//else if (restriction.EqualTo(substitution)) // Read or ("repeat" the state) // Each
//{
//    // No need to collect links to list
//    // Skip == Continue
//    // No need to check substituedHandler
//    if (!Memory.Each(link => !Equals(matchedHandler(Memory.GetLinkValue(link)),
↪  Constants.Break), restriction))
//        return Constants.Break;
//}
//else // Update
```

```
147             //{
148             //    //List<IList<T>> matchedLinks = null;
149             //    if (matchedHandler != null)
150             //    {
151             //        matchedLinks = new List<IList<T>>();
152             //        Func<T, bool> handler = link =>
153             //        {
154             //            var matchedLink = Memory.GetLinkValue(link);
155             //            var matchDecision = matchedHandler(matchedLink);
156             //            if (Equals(matchDecision, Constants.Break))
157             //                return false;
158             //            if (!Equals(matchDecision, Constants.Skip))
159             //                matchedLinks.Add(matchedLink);
160             //            return true;
161             //        };
162             //        if (!Memory.Each(handler, restriction))
163             //            return Constants.Break;
164             //    }
165             //    if (!matchedLinks.IsNullOrEmpty())
166             //    {
167             //        var totalMatchedLinks = matchedLinks.Count;
168             //        for (var i = 0; i < totalMatchedLinks; i++)
169             //        {
170             //            var matchedLink = matchedLinks[i];
171             //            if (substitutedHandler != null)
172             //            {
173             //                var newValue = new List<T>(); // TODO: Prepare value to update here
174             //                // TODO: Decide is it actually needed to use Before and After
                ↪  substitution handling.
175             //                var substitutedDecision = substitutedHandler(matchedLink,
                ↪  newValue);
176             //                if (Equals(substitutedDecision, Constants.Break))
177             //                    return Constants.Break;
178             //                if (Equals(substitutedDecision, Constants.Continue))
179             //                {
180             //                    // Actual update here
181             //                    Memory.SetLinkValue(newValue);
182             //                }
183             //                if (Equals(substitutedDecision, Constants.Skip))
184             //                {
185             //                    // Cancel the update. TODO: decide use separate Cancel
                ↪  constant or Skip is enough?
186             //                }
187             //            }
188             //        }
189             //    }
190             //}
191             return Constants.Continue;
192         }
193 
194         public TLink Trigger(IList<TLink> patternOrCondition, Func<IList<TLink>, TLink>
            ↪  matchHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
            ↪  substitutionHandler)
195         {
196             if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
197             {
198                 return Constants.Continue;
199             }
200             else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO:
                ↪  Check if it is a correct condition
201             {
202                 // Or it only applies to trigger without matchHandler.
203                 throw new NotImplementedException();
204             }
205             else if (!substitution.IsNullOrEmpty()) // Creation
206             {
207                 var before = ArrayPool<TLink>.Empty;
208                 // Что должно означать False здесь? Остановиться (перестать идти) или пропустить
                    ↪  (пройти мимо) или пустить (взять)?
209                 if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                    ↪  Constants.Break))
210                 {
211                     return Constants.Break;
212                 }
213                 var after = (IList<TLink>)substitution.ToArray();
214                 if (_equalityComparer.Equals(after[0], default))
215                 {
216                     var newLink = Links.Create();
```

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

```
                    if (matchHandler != null)
                    {
                        return substitutionHandler(before, after);
                    }
                    return Constants.Continue;
                }
                else
                {
                    throw new NotSupportedException();
                }
            }
        }

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

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

./Platform.Data.Doublets/DoubletComparer.cs

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

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

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

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

./Platform.Data.Doublets/Doublet.cs

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

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

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

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

```
            Target = target;
        }

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

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

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

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

./Platform.Data.Doublets/Hybrid.cs

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

```
60          public static explicit operator uint(Hybrid<T> hybrid) => Convert.ToUInt32(hybrid.Value);
61
62          public static explicit operator int(Hybrid<T> hybrid) =>
            ↪  Convert.ToInt32(hybrid.AbsoluteValue);
63
64          public static explicit operator ushort(Hybrid<T> hybrid) =>
            ↪  Convert.ToUInt16(hybrid.Value);
65
66          public static explicit operator short(Hybrid<T> hybrid) =>
            ↪  Convert.ToInt16(hybrid.AbsoluteValue);
67
68          public static explicit operator byte(Hybrid<T> hybrid) => Convert.ToByte(hybrid.Value);
69
70          public static explicit operator sbyte(Hybrid<T> hybrid) =>
            ↪  Convert.ToSByte(hybrid.AbsoluteValue);
71
72          public override string ToString() => IsNothing ? default(T) == null ? "Nothing" :
            ↪  default(T).ToString() : IsExternal ? $"<{AbsoluteValue}>" : Value.ToString();
73      }
74  }
```

./Platform.Data.Doublets/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   }
```

./Platform.Data.Doublets/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.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              {
```

```
            var linksAddressRange = new Range<ulong>(min, (Integer<TLink>)links.Count());
            Integer<TLink> link = RandomHelpers.Default.NextUInt64(linksAddressRange);
            links.Delete(link);
            if ((Integer<TLink>)links.Count() < min)
            {
                break;
            }
        }
    }

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

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

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

    #region Paths

    /// <remarks>
    /// TODO: Как так? Как то что ниже может быть корректно?
    /// Скорее всего практически не применимо
    /// Предполагалось, что можно было конвертировать формируемый в проходе через
    ↪  SequenceWalker
    /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
    /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
    /// </remarks>
    public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[]
    ↪  path)
    {
        var current = path[0];
        //EnsureLinkExists(current, "path");
        if (!links.Exists(current))
        {
            return false;
        }
        var equalityComparer = EqualityComparer<TLink>.Default;
```

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

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

```
244             var constants = links.Constants;
245             return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
                ↪  constants.Break, constants.Any, source, target);
246         }
247 
248         /// <summary>
249         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪  (handler) для каждой подходящей связи.
250         /// </summary>
251         /// <param name="links">Хранилище связей.</param>
252         /// <param name="source">Значение, определяющее соответствующие шаблону связи.
            ↪  (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
            ↪  Constants.Any - любое начало, 1..∞ конкретное начало)</param>
253         /// <param name="target">Значение, определяющее соответствующие шаблону связи.
            ↪  (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
            ↪  Constants.Any - любой конец, 1..∞ конкретный конец)</param>
254         /// <param name="handler">Обработчик каждой подходящей связи.</param>
255         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪  случае.</returns>
256         [MethodImpl(MethodImplOptions.AggressiveInlining)]
257         public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
            ↪  Func<IList<TLink>, TLink> handler)
258         {
259             var constants = links.Constants;
260             return links.Each(handler, constants.Any, source, target);
261         }
262 
263         [MethodImpl(MethodImplOptions.AggressiveInlining)]
264         public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
            ↪  restrictions)
265         {
266             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 = System.Math.Min(max, (Integer<TLink>)constants.MaxPossibleIndex);
372                 var createdLinks = new List<TLink>();
373                 var equalityComparer = EqualityComparer<TLink>.Default;
374                 TLink createdLink = creator();
```

```
                while (!equalityComparer.Equals(createdLink, (Integer<TLink>)max))
                {
                    createdLinks.Add(createdLink);
                }
                for (var i = 0; i < createdLinks.Count; i++)
                {
                    if (!nonExistentAddresses.Contains((Integer<TLink>)createdLinks[i]))
                    {
                        links.Delete(createdLinks[i]);
                    }
                }
            }
        }

        #endregion

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

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

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

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

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

```
445             return links.Update(link, link, link);
446         }
447 
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                 {
```

```
                    var references = ArrayPool.Allocate<TLink>((long)totalReferences);
                    var referencesFiller = new ArrayFiller<TLink, TLink>(references,
                    ↪  links.Constants.Continue);
                    links.Each(referencesFiller.AddFirstAndReturnConstant, constants.Any,
                    ↪  linkIndex, constants.Any);
                    links.Each(referencesFiller.AddFirstAndReturnConstant, constants.Any,
                    ↪  constants.Any, linkIndex);
                    for (ulong i = 0; i < referencesAsSourceCount; i++)
                    {
                        var reference = references[i];
                        if (equalityComparer.Equals(reference, linkIndex))
                        {
                            continue;
                        }

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

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

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

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

namespace Platform.Data.Doublets.Incrementers
{
    public class FrequencyIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly TLink _frequencyMarker;
        private readonly TLink _unaryOne;
        private readonly IIncrementer<TLink> _unaryNumberIncrementer;

        public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
        ↪  IIncrementer<TLink> unaryNumberIncrementer)
            : base(links)
        {
            _frequencyMarker = frequencyMarker;
            _unaryOne = unaryOne;
            _unaryNumberIncrementer = unaryNumberIncrementer;
        }

        public TLink Increment(TLink frequency)
        {
            if (_equalityComparer.Equals(frequency, default))
            {
                return Links.GetOrCreate(_unaryOne, _frequencyMarker);
            }
            var source = Links.GetSource(frequency);
            var incrementedSource = _unaryNumberIncrementer.Increment(source);
            return Links.GetOrCreate(incrementedSource, _frequencyMarker);
        }
    }
}
```

./Platform.Data.Doublets/Incrementers/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 IPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
        private readonly IIncrementer<TLink> _frequencyIncrementer;

        public LinkFrequencyIncrementer(ILinks<TLink> links, IPropertyOperator<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);
        }
    }
}
```

./Platform.Data.Doublets/Incrementers/UnaryNumberIncrementer.cs

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

namespace Platform.Data.Doublets.Incrementers
{
    public class UnaryNumberIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer = EqualityComparer<TLink>.Default;

        private readonly TLink _unaryOne;

        public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) : base(links) => _unaryOne = unaryOne;

        public TLink Increment(TLink unaryNumber)
        {
            if (_equalityComparer.Equals(unaryNumber, _unaryOne))
            {
                return Links.GetOrCreate(_unaryOne, _unaryOne);
            }
            var source = Links.GetSource(unaryNumber);
            var target = Links.GetTarget(unaryNumber);
            if (_equalityComparer.Equals(source, target))
            {
                return Links.GetOrCreate(unaryNumber, _unaryOne);
            }
            else
            {
                return Links.GetOrCreate(source, Increment(target));
            }
        }
    }
}
```

./Platform.Data.Doublets/ISynchronizedLinks.cs

```
using Platform.Data.Constants;

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

```
    }
}
```

./Platform.Data.Doublets/Link.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using Platform.Exceptions;
using Platform.Ranges;
using Platform.Singletons;
using Platform.Collections.Lists;
using Platform.Data.Constants;

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

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

        private const int Length = 3;

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

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

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

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

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

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

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

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

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

        public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
```

```
                                                && _equalityComparer.Equals(Source, other.Source)
                                                && _equalityComparer.Equals(Target, other.Target);

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

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

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

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

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

        #region IList

        public int Count => Length;

        public bool IsReadOnly => true;

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

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

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

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

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

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

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

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

        public int IndexOf(TLink item)
        {
```

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

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

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

        #endregion
    }
}
```

./Platform.Data.Doublets/LinkExtensions.cs

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

./Platform.Data.Doublets/LinksOperatorBase.cs

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

./Platform.Data.Doublets/PropertyOperators/DefaultLinkPropertyOperator.cs

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

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

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

        public TLink GetValue(TLink @object, TLink property)
        {
            var objectProperty = Links.SearchOrDefault(@object, property);
            if (_equalityComparer.Equals(objectProperty, default))
            {
                return default;
            }
            var valueLink = Links.All(Links.Constants.Any, objectProperty).SingleOrDefault();
            if (valueLink == null)
            {
                return default;
            }
            var value = Links.GetTarget(valueLink[Links.Constants.IndexPart]);
            return value;
        }

        public void SetValue(TLink @object, TLink property, TLink value)
```

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

./Platform.Data.Doublets/PropertyOperators/FrequencyPropertyOperator.cs

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

namespace Platform.Data.Doublets.PropertyOperators
{
    public class FrequencyPropertyOperator<TLink> : LinksOperatorBase<TLink>,
    ↪  IPropertyOperator<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);
            }
        }
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using Platform.Disposables;
using Platform.Singletons;
using Platform.Collections.Arrays;
using Platform.Numbers;
using Platform.Unsafe;
using Platform.Memory;
using Platform.Data.Exceptions;
using Platform.Data.Constants;
using static Platform.Numbers.Arithmetic;

#pragma warning disable 0649
#pragma warning disable 169
#pragma warning disable 618

// ReSharper disable StaticMemberInGenericType
// ReSharper disable BuiltInTypeReferenceStyle
// ReSharper disable MemberCanBePrivate.Local
// ReSharper disable UnusedMember.Local

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    public partial class ResizableDirectMemoryLinks<TLink> : DisposableBase, ILinks<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer = EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        /// <summary>Возвращает размер одной связи в байтах.</summary>
        public static readonly int LinkSizeInBytes = Structure<Link>.Size;

        public static readonly int LinkHeaderSizeInBytes = Structure<LinksHeader>.Size;

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

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

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetSource(IntPtr pointer) => (pointer + SourceOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetTarget(IntPtr pointer) => (pointer + TargetOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetLeftAsSource(IntPtr pointer) => (pointer + LeftAsSourceOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TLink GetRightAsSource(IntPtr pointer) => (pointer + RightAsSourceOffset).GetValue<TLink>();
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
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                 {
```

```
                    if (Exists(link) && _equalityComparer.Equals(handler(GetLinkStruct(link)),
                    ↪  Constants.Break))
                    {
                        return Constants.Break;
                    }
                }

                return Constants.Continue;
            }
            if (restrictions.Count == 1)
            {
                var index = restrictions[Constants.IndexPart];
                if (_equalityComparer.Equals(index, Constants.Any))
                {
                    return Each(handler, ArrayPool<TLink>.Empty);
                }
                if (!Exists(index))
                {
                    return Constants.Continue;
                }
                return handler(GetLinkStruct(index));
            }
            if (restrictions.Count == 2)
            {
                var index = restrictions[Constants.IndexPart];
                var value = restrictions[1];
                if (_equalityComparer.Equals(index, Constants.Any))
                {
                    if (_equalityComparer.Equals(value, Constants.Any))
                    {
                        return Each(handler, ArrayPool<TLink>.Empty);
                    }
                    if (_equalityComparer.Equals(Each(handler, new[] { index, value,
                    ↪  Constants.Any }), Constants.Break))
                    {
                        return Constants.Break;
                    }
                    return Each(handler, new[] { index, Constants.Any, value });
                }
                else
                {
                    if (!Exists(index))
                    {
                        return Constants.Continue;
                    }
                    if (_equalityComparer.Equals(value, Constants.Any))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    var storedLinkValue = GetLinkUnsafe(index);
                    if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
                        _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    return Constants.Continue;
                }
            }
            if (restrictions.Count == 3)
            {
                var index = restrictions[Constants.IndexPart];
                var source = restrictions[Constants.SourcePart];
                var target = restrictions[Constants.TargetPart];
                if (_equalityComparer.Equals(index, Constants.Any))
                {
                    if (_equalityComparer.Equals(source, Constants.Any) &&
                    ↪  _equalityComparer.Equals(target, Constants.Any))
                    {
                        return Each(handler, ArrayPool<TLink>.Empty);
                    }
                    else if (_equalityComparer.Equals(source, Constants.Any))
                    {
                        return _targetsTreeMethods.EachReference(target, handler);
                    }
                    else if (_equalityComparer.Equals(target, Constants.Any))
                    {
                        return _sourcesTreeMethods.EachReference(source, handler);
                    }
```

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

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

```
468             }
469             if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
470             {
471                 _targetsTreeMethods.Attach(LinksHeader.GetFirstAsTargetPointer(_header),
                    ↪ linkIndex);
472             }
473             return linkIndex;
474         }
475 
476         [MethodImpl(MethodImplOptions.AggressiveInlining)]
477         public Link<TLink> GetLinkStruct(TLink linkIndex)
478         {
479             var link = GetLinkUnsafe(linkIndex);
480             return new Link<TLink>(linkIndex, Link.GetSource(link), Link.GetTarget(link));
481         }
482 
483         [MethodImpl(MethodImplOptions.AggressiveInlining)]
484         private IntPtr GetLinkUnsafe(TLink linkIndex) => _links.GetElement(LinkSizeInBytes,
            ↪ linkIndex);
485 
486         /// <remarks>
487         /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
            ↪ пространство
488         /// </remarks>
489         public TLink Create()
490         {
491             var freeLink = LinksHeader.GetFirstFreeLink(_header);
492             if (!_equalityComparer.Equals(freeLink, Constants.Null))
493             {
494                 _unusedLinksListMethods.Detach(freeLink);
495             }
496             else
497             {
498                 if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
                    ↪ Constants.MaxPossibleIndex) > 0)
499                 {
500                     throw new
                        ↪ LinksLimitReachedException((Integer<TLink>)Constants.MaxPossibleIndex);
501                 }
502                 if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
                    ↪ Decrement(LinksHeader.GetReservedLinks(_header))) >= 0)
503                 {
504                     _memory.ReservedCapacity += _memoryReservationStep;
505                     SetPointers(_memory);
506                     LinksHeader.SetReservedLinks(_header,
                        ↪ (Integer<TLink>)(_memory.ReservedCapacity / LinkSizeInBytes));
507                 }
508                 LinksHeader.SetAllocatedLinks(_header,
                    ↪ Increment(LinksHeader.GetAllocatedLinks(_header)));
509                 _memory.UsedCapacity += LinkSizeInBytes;
510                 freeLink = LinksHeader.GetAllocatedLinks(_header);
511             }
512             return freeLink;
513         }
514 
515         public void Delete(TLink link)
516         {
517             if (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) < 0)
518             {
519                 _unusedLinksListMethods.AttachAsFirst(link);
520             }
521             else if (_equalityComparer.Equals(link, LinksHeader.GetAllocatedLinks(_header)))
522             {
523                 LinksHeader.SetAllocatedLinks(_header,
                    ↪ Decrement(LinksHeader.GetAllocatedLinks(_header)));
524                 _memory.UsedCapacity -= LinkSizeInBytes;
525                 // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
                    ↪ пока не дойдём до первой существующей связи
526                 // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
527                 while ((_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
                    ↪ Integer<TLink>.Zero) > 0) &&
                    ↪ IsUnusedLink(LinksHeader.GetAllocatedLinks(_header)))
528                 {
529                     _unusedLinksListMethods.Detach(LinksHeader.GetAllocatedLinks(_header));
530                     LinksHeader.SetAllocatedLinks(_header,
                        ↪ Decrement(LinksHeader.GetAllocatedLinks(_header)));
531                     _memory.UsedCapacity -= LinkSizeInBytes;
532                 }
```

```
            }
        }

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

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

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

        #region DisposableBase

        protected override bool AllowMultipleDisposeCalls => true;

        protected override void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                SetPointers(null);
                _memory.DisposeIfPossible();
            }
        }

        #endregion
    }
}
```

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

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

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;
            }
```

```
19
20              protected override TLink GetFirst() => (_header +
                ↪ LinksHeader.FirstFreeLinkOffset).GetValue<TLink>();
21
22              protected override TLink GetLast() => (_header +
                ↪ LinksHeader.LastFreeLinkOffset).GetValue<TLink>();
23
24              protected override TLink GetPrevious(TLink element) =>
                ↪ (_links.GetElement(LinkSizeInBytes, element) +
                ↪ Link.SourceOffset).GetValue<TLink>();
25
26              protected override TLink GetNext(TLink element) =>
                ↪ (_links.GetElement(LinkSizeInBytes, element) +
                ↪ Link.TargetOffset).GetValue<TLink>();
27
28              protected override TLink GetSize() => (_header +
                ↪ LinksHeader.FreeLinksOffset).GetValue<TLink>();
29
30              protected override void SetFirst(TLink element) => (_header +
                ↪ LinksHeader.FirstFreeLinkOffset).SetValue(element);
31
32              protected override void SetLast(TLink element) => (_header +
                ↪ LinksHeader.LastFreeLinkOffset).SetValue(element);
33
34              protected override void SetPrevious(TLink element, TLink previous) =>
                ↪ (_links.GetElement(LinkSizeInBytes, element) +
                ↪ Link.SourceOffset).SetValue(previous);
35
36              protected override void SetNext(TLink element, TLink next) =>
                ↪ (_links.GetElement(LinkSizeInBytes, element) + Link.TargetOffset).SetValue(next);
37
38              protected override void SetSize(TLink size) => (_header +
                ↪ LinksHeader.FreeLinksOffset).SetValue(size);
39          }
40      }
41  }
```

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

```
1   using System;
2   using System.Text;
3   using System.Collections.Generic;
4   using System.Runtime.CompilerServices;
5   using Platform.Numbers;
6   using Platform.Unsafe;
7   using Platform.Collections.Methods.Trees;
8   using Platform.Data.Constants;
9
10  namespace Platform.Data.Doublets.ResizableDirectMemory
11  {
12      partial class ResizableDirectMemoryLinks<TLink>
13      {
14          private abstract class LinksTreeMethodsBase :
                ↪ SizedAndThreadedAVLBalancedTreeMethods<TLink>
15          {
16              private readonly ResizableDirectMemoryLinks<TLink> _memory;
17              private readonly LinksCombinedConstants<TLink, TLink, int> _constants;
18              protected readonly IntPtr Links;
19              protected readonly IntPtr Header;
20
21              protected LinksTreeMethodsBase(ResizableDirectMemoryLinks<TLink> memory)
22              {
23                  Links = memory._links;
24                  Header = memory._header;
25                  _memory = memory;
26                  _constants = memory.Constants;
27              }
28
29              [MethodImpl(MethodImplOptions.AggressiveInlining)]
30              protected abstract TLink GetTreeRoot();
31
32              [MethodImpl(MethodImplOptions.AggressiveInlining)]
33              protected abstract TLink GetBasePartValue(TLink link);
34
35              public TLink this[TLink index]
36              {
37                  get
38                  {
39                      var root = GetTreeRoot();
40                      if (GreaterOrEqualThan(index, GetSize(root)))
41                      {
```

```
                    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;
                    }
```

```
119                         current = GetLeftOrDefault(current);
120                     }
121                     else
122                     {
123                         current = GetRightOrDefault(current);
124                     }
125                 }
126                 if (!EqualToZero(first))
127                 {
128                     current = first;
129                     while (true)
130                     {
131                         if (IsEquals(handler(_memory.GetLinkStruct(current)), _constants.Break))
132                         {
133                             return _constants.Break;
134                         }
135                         current = GetNext(current);
136                         if (EqualToZero(current) || !IsEquals(GetBasePartValue(current), link))
137                         {
138                             break;
139                         }
140                     }
141                 }
142                 return _constants.Continue;
143             }
144 
145             protected override void PrintNodeValue(TLink node, StringBuilder sb)
146             {
147                 sb.Append(' ');
148                 sb.Append((Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SourceOffset).GetValue<TLink>());
149                 sb.Append('-');
150                 sb.Append('>');
151                 sb.Append((Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.TargetOffset).GetValue<TLink>());
152             }
153         }
154 
155         private class LinksSourcesTreeMethods : LinksTreeMethodsBase
156         {
157             public LinksSourcesTreeMethods(ResizableDirectMemoryLinks<TLink> memory)
158                 : base(memory)
159             {
160             }
161 
162             protected override IntPtr GetLeftPointer(TLink node) =>
                ↪ Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsSourceOffset;
163 
164             protected override IntPtr GetRightPointer(TLink node) =>
                ↪ Links.GetElement(LinkSizeInBytes, node) + Link.RightAsSourceOffset;
165 
166             protected override TLink GetLeftValue(TLink node) =>
                ↪ (Links.GetElement(LinkSizeInBytes, node) +
                ↪ Link.LeftAsSourceOffset).GetValue<TLink>();
167 
168             protected override TLink GetRightValue(TLink node) =>
                ↪ (Links.GetElement(LinkSizeInBytes, node) +
                ↪ Link.RightAsSourceOffset).GetValue<TLink>();
169 
170             protected override TLink GetSize(TLink node)
171             {
172                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
173                 return Bit.PartialRead(previousValue, 5, -5);
174             }
175 
176             protected override void SetLeft(TLink node, TLink left) =>
                ↪ (Links.GetElement(LinkSizeInBytes, node) +
                ↪ Link.LeftAsSourceOffset).SetValue(left);
177 
178             protected override void SetRight(TLink node, TLink right) =>
                ↪ (Links.GetElement(LinkSizeInBytes, node) +
                ↪ Link.RightAsSourceOffset).SetValue(right);
179 
180             protected override void SetSize(TLink node, TLink size)
181             {
182                 var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
                    ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
```

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

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

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

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

./Platform.Data.Doublets/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.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
19 {
20     using id = UInt64;
21
22     public unsafe partial class UInt64ResizableDirectMemoryLinks : DisposableBase, ILinks<id>
23     {
24         /// <summary>Возвращает размер одной связи в байтах.</summary>
25         /// <remarks>
26         /// Используется только во вне класса, не рекомедуется использовать внутри.
27         /// Так как во вне не обязательно будет доступен unsafe C#.
28         /// </remarks>
```

```
29          public static readonly int LinkSizeInBytes = sizeof(Link);
30
31          public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
32
33          private struct Link
34          {
35              public id Source;
36              public id Target;
37              public id LeftAsSource;
38              public id RightAsSource;
39              public id SizeAsSource;
40              public id LeftAsTarget;
41              public id RightAsTarget;
42              public id SizeAsTarget;
43          }
44
45          private struct LinksHeader
46          {
47              public id AllocatedLinks;
48              public id ReservedLinks;
49              public id FreeLinks;
50              public id FirstFreeLink;
51              public id FirstAsSource;
52              public id FirstAsTarget;
53              public id LastFreeLink;
54              public id Reserved8;
55          }
56
57          private readonly long _memoryReservationStep;
58
59          private readonly IResizableDirectMemory _memory;
60          private LinksHeader* _header;
61          private Link* _links;
62
63          private LinksTargetsTreeMethods _targetsTreeMethods;
64          private LinksSourcesTreeMethods _sourcesTreeMethods;
65
66          // 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
67          private UnusedLinksListMethods _unusedLinksListMethods;
68
69          /// <summary>
70          /// Возвращает общее число связей находящихся в хранилище.
71          /// </summary>
72          private id Total => _header->AllocatedLinks - _header->FreeLinks;
73
74          // TODO: Дать возможность переопределять в конструкторе
75          public LinksCombinedConstants<id, id, int> Constants { get; }
76
77          public UInt64ResizableDirectMemoryLinks(string address) : this(address,
            ↪  DefaultLinksSizeStep) { }
78
79          /// <summary>
80          /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
            ↪  минимальным шагом расширения базы данных.
81          /// </summary>
82          /// <param name="address">Полный пусть к файлу базы данных.</param>
83          /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
            ↪  байтах.</param>
84          public UInt64ResizableDirectMemoryLinks(string address, long memoryReservationStep) :
            ↪  this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
            ↪  memoryReservationStep) { }
85
86          public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
            ↪  DefaultLinksSizeStep) { }
87
88          public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
            ↪  memoryReservationStep)
89          {
90              Constants = Default<LinksCombinedConstants<id, id, int>>.Instance;
91              _memory = memory;
92              _memoryReservationStep = memoryReservationStep;
93              if (memory.ReservedCapacity < memoryReservationStep)
94              {
95                  memory.ReservedCapacity = memoryReservationStep;
96              }
97              SetPointers(_memory);
98              // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
99              _memory.UsedCapacity = ((long)_header->AllocatedLinks * sizeof(Link)) +
                ↪  sizeof(LinksHeader);
```

```
100             // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
101             _header->ReservedLinks = (id)((_memory.ReservedCapacity - sizeof(LinksHeader)) /
                ↪  sizeof(Link));
102         }
103
104         [MethodImpl(MethodImplOptions.AggressiveInlining)]
105         public id Count(IList<id> restrictions)
106         {
107             // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
108             if (restrictions.Count == 0)
109             {
110                 return Total;
111             }
112             if (restrictions.Count == 1)
113             {
114                 var index = restrictions[Constants.IndexPart];
115                 if (index == Constants.Any)
116                 {
117                     return Total;
118                 }
119                 return Exists(index) ? 1UL : 0UL;
120             }
121             if (restrictions.Count == 2)
122             {
123                 var index = restrictions[Constants.IndexPart];
124                 var value = restrictions[1];
125                 if (index == Constants.Any)
126                 {
127                     if (value == Constants.Any)
128                     {
129                         return Total; // Any - как отсутствие ограничения
130                     }
131                     return _sourcesTreeMethods.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);
167                     }
168                     else if (target == Constants.Any)
169                     {
170                         return _sourcesTreeMethods.CalculateReferences(source);
171                     }
172                     else //if(source != Any && target != Any)
173                     {
174                         // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
175                         var link = _sourcesTreeMethods.Search(source, target);
176                         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)
                {
```

```
255                     if (value == Constants.Any)
256                     {
257                         return Each(handler, ArrayPool<ulong>.Empty);
258                     }
259                     if (Each(handler, new[] { index, value, Constants.Any }) == Constants.Break)
260                     {
261                         return Constants.Break;
262                     }
263                     return Each(handler, new[] { index, Constants.Any, value });
264                 }
265                 else
266                 {
267                     if (!Exists(index))
268                     {
269                         return Constants.Continue;
270                     }
271                     if (value == Constants.Any)
272                     {
273                         return handler(GetLinkStruct(index));
274                     }
275                     var storedLinkValue = GetLinkUnsafe(index);
276                     if (storedLinkValue->Source == value ||
277                         storedLinkValue->Target == value)
278                     {
279                         return handler(GetLinkStruct(index));
280                     }
281                     return Constants.Continue;
282                 }
283             }
284             if (restrictions.Count == 3)
285             {
286                 var index = restrictions[Constants.IndexPart];
287                 var source = restrictions[Constants.SourcePart];
288                 var target = restrictions[Constants.TargetPart];
289                 if (index == Constants.Any)
290                 {
291                     if (source == Constants.Any && target == Constants.Any)
292                     {
293                         return Each(handler, ArrayPool<ulong>.Empty);
294                     }
295                     else if (source == Constants.Any)
296                     {
297                         return _targetsTreeMethods.EachReference(target, handler);
298                     }
299                     else if (target == Constants.Any)
300                     {
301                         return _sourcesTreeMethods.EachReference(source, handler);
302                     }
303                     else //if(source != Any && target != Any)
304                     {
305                         var link = _sourcesTreeMethods.Search(source, target);
306                         return link == Constants.Null ? Constants.Continue :
                            ↪ handler(GetLinkStruct(link));
307                     }
308                 }
309                 else
310                 {
311                     if (!Exists(index))
312                     {
313                         return Constants.Continue;
314                     }
315                     if (source == Constants.Any && target == Constants.Any)
316                     {
317                         return handler(GetLinkStruct(index));
318                     }
319                     var storedLinkValue = GetLinkUnsafe(index);
320                     if (source != Constants.Any && target != Constants.Any)
321                     {
322                         if (storedLinkValue->Source == source &&
323                             storedLinkValue->Target == target)
324                         {
325                             return handler(GetLinkStruct(index));
326                         }
327                         return Constants.Continue;
328                     }
329                     var value = default(id);
330                     if (source == Constants.Any)
331                     {
```

```
                        value = target;
                    }
                    if (target == Constants.Any)
                    {
                        value = source;
                    }
                    if (storedLinkValue->Source == value ||
                        storedLinkValue->Target == value)
                    {
                        return handler(GetLinkStruct(index));
                    }
                    return Constants.Continue;
                }
            }
            throw new NotSupportedException("Другие размеры и способы ограничений не
            ↪  поддерживаются.");
        }

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

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

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

        /// <remarks>
        /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
        ↪  пространство
```

```
        /// </remarks>
        public id Create()
        {
            var freeLink = _header->FirstFreeLink;
            if (freeLink != Constants.Null)
            {
                _unusedLinksListMethods.Detach(freeLink);
            }
            else
            {
                if (_header->AllocatedLinks > Constants.MaxPossibleIndex)
                {
                    throw new LinksLimitReachedException(Constants.MaxPossibleIndex);
                }
                if (_header->AllocatedLinks >= _header->ReservedLinks - 1)
                {
                    _memory.ReservedCapacity += _memoryReservationStep;
                    SetPointers(_memory);
                    _header->ReservedLinks = (id)(_memory.ReservedCapacity / sizeof(Link));
                }
                _header->AllocatedLinks++;
                _memory.UsedCapacity += sizeof(Link);
                freeLink = _header->AllocatedLinks;
            }
            return freeLink;
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        private bool Exists(id link) => link >= Constants.MinPossibleIndex && link <=
        ↪  _header->AllocatedLinks && !IsUnusedLink(link);

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

        #region Disposable

        protected override bool AllowMultipleDisposeCalls => true;

        protected override void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                SetPointers(null);
                _memory.DisposeIfPossible();
            }
        }

        #endregion
    }
}
```

./Platform.Data.Doublets/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;
        }
    }
}
```

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

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

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    unsafe partial class UInt64ResizableDirectMemoryLinks
    {
```

```
12          private abstract class LinksTreeMethodsBase :
            ↪ SizedAndThreadedAVLBalancedTreeMethods<ulong>
13          {
14              private readonly UInt64ResizableDirectMemoryLinks _memory;
15              private readonly LinksCombinedConstants<ulong, ulong, int> _constants;
16              protected readonly Link* Links;
17              protected readonly LinksHeader* Header;
18
19              protected LinksTreeMethodsBase(UInt64ResizableDirectMemoryLinks memory)
20              {
21                  Links = memory._links;
22                  Header = memory._header;
23                  _memory = memory;
24                  _constants = memory.Constants;
25              }
26
27              [MethodImpl(MethodImplOptions.AggressiveInlining)]
28              protected abstract ulong GetTreeRoot();
29
30              [MethodImpl(MethodImplOptions.AggressiveInlining)]
31              protected abstract ulong GetBasePartValue(ulong link);
32
33              public ulong this[ulong index]
34              {
35                  get
36                  {
37                      var root = GetTreeRoot();
38                      if (index >= GetSize(root))
39                      {
40                          return 0;
41                      }
42                      while (root != 0)
43                      {
44                          var left = GetLeftOrDefault(root);
45                          var leftSize = GetSizeOrZero(left);
46                          if (index < leftSize)
47                          {
48                              root = left;
49                              continue;
50                          }
51                          if (index == leftSize)
52                          {
53                              return root;
54                          }
55                          root = GetRightOrDefault(root);
56                          index -= leftSize + 1;
57                      }
58                      return 0; // TODO: Impossible situation exception (only if tree structure
                        ↪ broken)
59                  }
60              }
61
62              // TODO: Return indices range instead of references count
63              public ulong CalculateReferences(ulong link)
64              {
65                  var root = GetTreeRoot();
66                  var total = GetSize(root);
67                  var totalRightIgnore = 0UL;
68                  while (root != 0)
69                  {
70                      var @base = GetBasePartValue(root);
71                      if (@base <= link)
72                      {
73                          root = GetRightOrDefault(root);
74                      }
75                      else
76                      {
77                          totalRightIgnore += GetRightSize(root) + 1;
78                          root = GetLeftOrDefault(root);
79                      }
80                  }
81                  root = GetTreeRoot();
82                  var totalLeftIgnore = 0UL;
83                  while (root != 0)
84                  {
85                      var @base = GetBasePartValue(root);
86                      if (@base >= link)
87                      {
88                          root = GetLeftOrDefault(root);
```

```
                }
                else
                {
                    totalLeftIgnore += GetLeftSize(root) + 1;
                    root = GetRightOrDefault(root);
                }
            }
            return total - totalRightIgnore - totalLeftIgnore;
        }

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

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

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

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

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

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

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

```
166
167             protected override ulong GetSize(ulong node)
168             {
169                 var previousValue = Links[node].SizeAsSource;
170                 //return Math.PartialRead(previousValue, 5, -5);
171                 return (previousValue & 4294967264) >> 5;
172             }
173
174             protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource
                ↪  = left;
175
176             protected override void SetRight(ulong node, ulong right) =>
                ↪  Links[node].RightAsSource = right;
177
178             protected override void SetSize(ulong node, ulong size)
179             {
180                 var previousValue = Links[node].SizeAsSource;
181                 //var modified = Math.PartialWrite(previousValue, size, 5, -5);
182                 var modified = (previousValue & 31) | ((size & 134217727) << 5);
183                 Links[node].SizeAsSource = modified;
184             }
185
186             protected override bool GetLeftIsChild(ulong node)
187             {
188                 var previousValue = Links[node].SizeAsSource;
189                 //return (Integer)Math.PartialRead(previousValue, 4, 1);
190                 return (previousValue & 16) >> 4 == 1UL;
191             }
192
193             protected override void SetLeftIsChild(ulong node, bool value)
194             {
195                 var previousValue = Links[node].SizeAsSource;
196                 //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 4, 1);
197                 var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
198                 Links[node].SizeAsSource = modified;
199             }
200
201             protected override bool GetRightIsChild(ulong node)
202             {
203                 var previousValue = Links[node].SizeAsSource;
204                 //return (Integer)Math.PartialRead(previousValue, 3, 1);
205                 return (previousValue & 8) >> 3 == 1UL;
206             }
207
208             protected override void SetRightIsChild(ulong node, bool value)
209             {
210                 var previousValue = Links[node].SizeAsSource;
211                 //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 3, 1);
212                 var modified = (previousValue & 4294967287) | ((value ? 1UL : 0UL) << 3);
213                 Links[node].SizeAsSource = modified;
214             }
215
216             protected override sbyte GetBalance(ulong node)
217             {
218                 var previousValue = Links[node].SizeAsSource;
219                 //var value = Math.PartialRead(previousValue, 0, 3);
220                 var value = previousValue & 7;
221                 var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
                    ↪  124 : value & 3);
222                 return unpackedValue;
223             }
224
225             protected override void SetBalance(ulong node, sbyte value)
226             {
227                 var previousValue = Links[node].SizeAsSource;
228                 var packagedValue = (ulong)((((byte)value >> 5) & 4) | value & 3);
229                 //var modified = Math.PartialWrite(previousValue, packagedValue, 0, 3);
230                 var modified = (previousValue & 4294967288) | (packagedValue & 7);
231                 Links[node].SizeAsSource = modified;
232             }
233
234             protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
235                 => Links[first].Source < Links[second].Source ||
236                   (Links[first].Source == Links[second].Source && Links[first].Target <
                    ↪  Links[second].Target);
237
238             protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
239                 => Links[first].Source > Links[second].Source ||
```

```
            (Links[first].Source == Links[second].Source && Links[first].Target > Links[second].Target);

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

```
374             protected override ulong GetSize(ulong node)
375             {
376                 var previousValue = Links[node].SizeAsTarget;
377                 //return Math.PartialRead(previousValue, 5, -5);
378                 return (previousValue & 4294967264) >> 5;
379             }
380 
381             protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget
                ↪  = left;
382 
383             protected override void SetRight(ulong node, ulong right) =>
                ↪  Links[node].RightAsTarget = right;
384 
385             protected override void SetSize(ulong node, ulong size)
386             {
387                 var previousValue = Links[node].SizeAsTarget;
388                 //var modified = Math.PartialWrite(previousValue, size, 5, -5);
389                 var modified = (previousValue & 31) | ((size & 134217727) << 5);
390                 Links[node].SizeAsTarget = modified;
391             }
392 
393             protected override bool GetLeftIsChild(ulong node)
394             {
395                 var previousValue = Links[node].SizeAsTarget;
396                 //return (Integer)Math.PartialRead(previousValue, 4, 1);
397                 return (previousValue & 16) >> 4 == 1UL;
398                 // TODO: Check if this is possible to use
399                 //var nodeSize = GetSize(node);
400                 //var left = GetLeftValue(node);
401                 //var leftSize = GetSizeOrZero(left);
402                 //return leftSize > 0 && nodeSize > leftSize;
403             }
404 
405             protected override void SetLeftIsChild(ulong node, bool value)
406             {
407                 var previousValue = Links[node].SizeAsTarget;
408                 //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 4, 1);
409                 var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
410                 Links[node].SizeAsTarget = modified;
411             }
412 
413             protected override bool GetRightIsChild(ulong node)
414             {
415                 var previousValue = Links[node].SizeAsTarget;
416                 //return (Integer)Math.PartialRead(previousValue, 3, 1);
417                 return (previousValue & 8) >> 3 == 1UL;
418                 // TODO: Check if this is possible to use
419                 //var nodeSize = GetSize(node);
420                 //var right = GetRightValue(node);
421                 //var rightSize = GetSizeOrZero(right);
422                 //return rightSize > 0 && nodeSize > rightSize;
423             }
424 
425             protected override void SetRightIsChild(ulong node, bool value)
426             {
427                 var previousValue = Links[node].SizeAsTarget;
428                 //var modified = Math.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             {
435                 var previousValue = Links[node].SizeAsTarget;
436                 //var value = Math.PartialRead(previousValue, 0, 3);
437                 var value = previousValue & 7;
438                 var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
                    ↪  124 : value & 3);
439                 return unpackedValue;
440             }
441 
442             protected override void SetBalance(ulong node, sbyte value)
443             {
444                 var previousValue = Links[node].SizeAsTarget;
445                 var packagedValue = (ulong)((((byte)value >> 5) & 4) | value & 3);
446                 //var modified = Math.PartialWrite(previousValue, packagedValue, 0, 3);
447                 var modified = (previousValue & 4294967288) | (packagedValue & 7);
448                 Links[node].SizeAsTarget = modified;
```

```
            }

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

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

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

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

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

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

```
using System.Collections.Generic;

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

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

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

```
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Collections;
using Platform.Singletons;
using Platform.Numbers;
using Platform.Data.Constants;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;

namespace Platform.Data.Doublets.Sequences.Converters
{
    /// <remarks>
    /// TODO: Возможно будет лучше если алгоритм будет выполняться полностью изолированно от Links на этапе сжатия.
    ///     А именно будет создаваться временный список пар необходимых для выполнения сжатия, в таком случае тип значения элемента массива может быть любым, как char так и ulong.
    ///     Как только список/словарь пар был выявлен можно разом выполнить создание всех этих пар, а так же разом выполнить замену.
    /// </remarks>
    public class CompressingConverter<TLink> : LinksListToSequenceConverterBase<TLink>
    {
        private static readonly LinksCombinedConstants<bool, TLink, long> _constants = Default<LinksCombinedConstants<bool, TLink, long>>.Instance;
        private static readonly EqualityComparer<TLink> _equalityComparer = EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        private readonly IConverter<IList<TLink>, TLink> _baseConverter;
        private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
        private readonly TLink _minFrequencyToCompress;
        private readonly bool _doInitialFrequenciesIncrement;
        private Doublet<TLink> _maxDoublet;
        private LinkFrequency<TLink> _maxDoubletData;

        private struct HalfDoublet
        {
            public TLink Element;
            public LinkFrequency<TLink> DoubletData;

            public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
            {
                Element = element;
                DoubletData = doubletData;
            }

            public override string ToString() => $"{Element}: ({DoubletData})";
        }

        public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink> baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
            : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One, true)
        {
        }

        public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink> baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool doInitialFrequenciesIncrement)
            : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One, doInitialFrequenciesIncrement)
        {
        }

        public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink> baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink minFrequencyToCompress, bool doInitialFrequenciesIncrement)
            : base(links)
        {
            _baseConverter = baseConverter;
            _doubletFrequenciesCache = doubletFrequenciesCache;
            if (_comparer.Compare(minFrequencyToCompress, Integer<TLink>.One) < 0)
            {
                minFrequencyToCompress = Integer<TLink>.One;
            }
            _minFrequencyToCompress = minFrequencyToCompress;
            _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
            ResetMaxDoublet();
```

```
        }

        public override TLink Convert(IList<TLink> source) =>
        ↪  _baseConverter.Convert(Compress(source));

        /// <remarks>
        /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding .
        /// Faster version (doublets' frequencies dictionary is not recreated).
        /// </remarks>
        private IList<TLink> Compress(IList<TLink> sequence)
        {
            if (sequence.IsNullOrEmpty())
            {
                return null;
            }
            if (sequence.Count == 1)
            {
                return sequence;
            }
            if (sequence.Count == 2)
            {
                return new[] { Links.GetOrCreate(sequence[0], sequence[1]) };
            }
            // TODO: arraypool with min size (to improve cache locality) or stackallow with Sigil
            var copy = new HalfDoublet[sequence.Count];
            Doublet<TLink> doublet = default;
            for (var i = 1; i < sequence.Count; i++)
            {
                doublet.Source = sequence[i - 1];
                doublet.Target = sequence[i];
                LinkFrequency<TLink> data;
                if (_doInitialFrequenciesIncrement)
                {
                    data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
                }
                else
                {
                    data = _doubletFrequenciesCache.GetFrequency(ref doublet);
                    if (data == null)
                    {
                        throw new NotSupportedException("If you ask not to increment
                        ↪  frequencies, it is expected that all frequencies for the sequence
                        ↪  are prepared.");
                    }
                }
                copy[i - 1].Element = sequence[i - 1];
                copy[i - 1].DoubletData = data;
                UpdateMaxDoublet(ref doublet, data);
            }
            copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
            copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
            if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
            {
                var newLength = ReplaceDoublets(copy);
                sequence = new TLink[newLength];
                for (int i = 0; i < newLength; i++)
                {
                    sequence[i] = copy[i].Element;
                }
            }
            return sequence;
        }

        /// <remarks>
        /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding
        /// </remarks>
        private int ReplaceDoublets(HalfDoublet[] copy)
        {
            var oldLength = copy.Length;
            var newLength = copy.Length;
            while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
            {
                var maxDoubletSource = _maxDoublet.Source;
                var maxDoubletTarget = _maxDoublet.Target;
                if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
                {
                    _maxDoubletData.Link = Links.GetOrCreate(maxDoubletSource, maxDoubletTarget);
                }
                var maxDoubletReplacementLink = _maxDoubletData.Link;
```

```
                oldLength--;
                var oldLengthMinusTwo = oldLength - 1;
                // Substitute all usages
                int w = 0, r = 0; // (r == read, w == write)
                for (; r < oldLength; r++)
                {
                    if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
                    ↪  _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
                    {
                        if (r > 0)
                        {
                            var previous = copy[w - 1].Element;
                            copy[w - 1].DoubletData.DecrementFrequency();
                            copy[w - 1].DoubletData =
                            ↪  _doubletFrequenciesCache.IncrementFrequency(previous,
                            ↪  maxDoubletReplacementLink);
                        }
                        if (r < oldLengthMinusTwo)
                        {
                            var next = copy[r + 2].Element;
                            copy[r + 1].DoubletData.DecrementFrequency();
                            copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(ma
                            ↪  xDoubletReplacementLink,
                            ↪  next);
                        }
                        copy[w++].Element = maxDoubletReplacementLink;
                        r++;
                        newLength--;
                    }
                    else
                    {
                        copy[w++] = copy[r];
                    }
                }
                if (w < newLength)
                {
                    copy[w] = copy[r];
                }
                oldLength = newLength;
                ResetMaxDoublet();
                UpdateMaxDoublet(copy, newLength);
            }
            return newLength;
        }

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

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

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

```
209                 (_comparer.Compare(maxFrequency, frequency) < 0 ||
                        (_equalityComparer.Equals(maxFrequency, frequency) &&
                        _comparer.Compare(Arithmetic.Add(doublet.Source, doublet.Target),
                        Arithmetic.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /* gives
                        better stability and better compression on sequent data and even on rundom
                        numbers data (but gives collisions anyway) */
210             {
211                 _maxDoublet = doublet;
212                 _maxDoubletData = data;
213             }
214         }
215     }
216 }
```

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

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

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

```
1  using System.Collections.Generic;
2  using System.Linq;
3  using Platform.Interfaces;
4
5  namespace Platform.Data.Doublets.Sequences.Converters
6  {
7      public class OptimalVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
8      {
9          private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
10         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
11
12         private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;
13
14         public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
           ↪  sequenceToItsLocalElementLevelsConverter) : base(links)
15             => _sequenceToItsLocalElementLevelsConverter =
                ↪  sequenceToItsLocalElementLevelsConverter;
16
17         public override TLink Convert(IList<TLink> sequence)
18         {
19             var length = sequence.Count;
20             if (length == 1)
21             {
22                 return sequence[0];
23             }
24             var links = Links;
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;
37                 var w = 0;
38                 for (var i = 1; i < length; i++)
39                 {
40                     if (_equalityComparer.Equals(currentLevel, levels[i]))
41                     {
42                         levelRepeat++;
43                         skipOnce = false;
44                         if (levelRepeat == 2)
45                         {
46                             sequence[w] = links.GetOrCreate(sequence[i - 1], sequence[i]);
```

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

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

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

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

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

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

namespace Platform.Data.Doublets.Sequences.Converters
{
    public class SequenceToItsLocalElementLevelsConverter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<IList<TLink>>
    {
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
        private readonly IConverter<Doublet<TLink>, TLink> _linkToItsFrequencyToNumberConveter;
        public SequenceToItsLocalElementLevelsConverter(ILinks<TLink> links,
        ↪ IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter) : base(links)
        ↪ => _linkToItsFrequencyToNumberConveter = linkToItsFrequencyToNumberConveter;
        public IList<TLink> Convert(IList<TLink> sequence)
```

```
        {
            var levels = new TLink[sequence.Count];
            levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
            for (var i = 1; i < sequence.Count - 1; i++)
            {
                var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
                var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
                levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
            }
            levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
            ↪  sequence[sequence.Count - 1]);
            return levels;
        }

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

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

```
using Platform.Interfaces;

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

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

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

namespace Platform.Data.Doublets.Sequences.CreteriaMatchers
{
    public class MarkedSequenceCreteriaMatcher<TLink> : ICriterionMatcher<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);
    }
}
```

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

```
using System.Collections.Generic;
using Platform.Collections.Stacks;
using Platform.Data.Doublets.Sequences.HeightProviders;
using Platform.Data.Sequences;

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

        private readonly IStack<TLink> _stack;
        private readonly ISequenceHeightProvider<TLink> _heightProvider;

        public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
        ↪  ISequenceHeightProvider<TLink> heightProvider)
            : base(links)
```

```
        {
            _stack = stack;
            _heightProvider = heightProvider;
        }

        public TLink Append(TLink sequence, TLink appendant)
        {
            var cursor = sequence;
            while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
            {
                var source = Links.GetSource(cursor);
                var target = Links.GetTarget(cursor);
                if (_equalityComparer.Equals(_heightProvider.Get(source),
                ↪  _heightProvider.Get(target)))
                {
                    break;
                }
                else
                {
                    _stack.Push(source);
                    cursor = target;
                }
            }
            var left = cursor;
            var right = appendant;
            while (!_equalityComparer.Equals(cursor = _stack.Pop(), Links.Constants.Null))
            {
                right = Links.GetOrCreate(left, right);
                left = cursor;
            }
            return Links.GetOrCreate(left, right);
        }
    }
}
```

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

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

namespace Platform.Data.Doublets.Sequences
{
    public class DuplicateSegmentsCounter<TLink> : ICounter<int>
    {
        private readonly IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
        ↪  _duplicateFragmentsProvider;
        public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLink>,
        ↪  IList<TLink>>>> duplicateFragmentsProvider) => _duplicateFragmentsProvider =
        ↪  duplicateFragmentsProvider;
        public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
    }
}
```

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

```
using System;
using System.Linq;
using System.Collections.Generic;
using Platform.Interfaces;
using Platform.Collections;
using Platform.Collections.Lists;
using Platform.Collections.Segments;
using Platform.Collections.Segments.Walkers;
using Platform.Singletons;
using Platform.Numbers;
using Platform.Data.Sequences;

namespace Platform.Data.Doublets.Sequences
{
    public class DuplicateSegmentsProvider<TLink> :
    ↪  DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
    ↪  IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
    {
        private readonly ILinks<TLink> _links;
        private readonly ISequences<TLink> _sequences;
        private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
        private BitString _visited;

        private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
        ↪  IList<TLink>>>
        {
```

```
24              private readonly IListEqualityComparer<TLink> _listComparer;
25              public ItemEquilityComparer() => _listComparer =
                ↪  Default<IListEqualityComparer<TLink>>.Instance;
26              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);
27              public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
                ↪  (_listComparer.GetHashCode(pair.Key),
                ↪  _listComparer.GetHashCode(pair.Value)).GetHashCode();
28          }
29
30          private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
31          {
32              private readonly IListComparer<TLink> _listComparer;
33
34              public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;
35
36              public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
                ↪  KeyValuePair<IList<TLink>, IList<TLink>> right)
37              {
38                  var intermediateResult = _listComparer.Compare(left.Key, right.Key);
39                  if (intermediateResult == 0)
40                  {
41                      intermediateResult = _listComparer.Compare(left.Value, right.Value);
42                  }
43                  return intermediateResult;
44              }
45          }
46
47          public DuplicateSegmentsProvider(ILinks<TLink> links, ISequences<TLink> sequences)
48              : base(minimumStringSegmentLength: 2)
49          {
50              _links = links;
51              _sequences = sequences;
52          }
53
54          public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
55          {
56              _groups = new HashSet<KeyValuePair<IList<TLink>,
                ↪  IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
57              var count = _links.Count();
58              _visited = new BitString((long)(Integer<TLink>)count + 1);
59              _links.Each(link =>
60              {
61                  var linkIndex = _links.GetIndex(link);
62                  var linkBitIndex = (long)(Integer<TLink>)linkIndex;
63                  if (!_visited.Get(linkBitIndex))
64                  {
65                      var sequenceElements = new List<TLink>();
66                      _sequences.EachPart(sequenceElements.AddAndReturnTrue, linkIndex);
67                      if (sequenceElements.Count > 2)
68                      {
69                          WalkAll(sequenceElements);
70                      }
71                  }
72                  return _links.Constants.Continue;
73              });
74              var resultList = _groups.ToList();
75              var comparer = Default<ItemComparer>.Instance;
76              resultList.Sort(comparer);
77  #if DEBUG
78              foreach (var item in resultList)
79              {
80                  PrintDuplicates(item);
81              }
82  #endif
83              return resultList;
84          }
85
86          protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
            ↪  length) => new Segment<TLink>(elements, offset, length);
87
88          protected override void OnDublicateFound(Segment<TLink> segment)
89          {
90              var duplicates = CollectDuplicatesForSegment(segment);
91              if (duplicates.Count > 1)
92              {
```

```
                    _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
                    ↪  duplicates));
                }
            }

            private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
            {
                var duplicates = new List<TLink>();
                var readAsElement = new HashSet<TLink>();
                _sequences.Each(sequence =>
                {
                    duplicates.Add(sequence);
                    readAsElement.Add(sequence);
                    return true; // Continue
                }, segment);
                if (duplicates.Any(x => _visited.Get((Integer<TLink>)x)))
                {
                    return new List<TLink>();
                }
                foreach (var duplicate in duplicates)
                {
                    var duplicateBitIndex = (long)(Integer<TLink>)duplicate;
                    _visited.Set(duplicateBitIndex);
                }
                if (_sequences is Sequences sequencesExperiments)
                {
                    var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((H
                    ↪  ashSet<ulong>)(object)readAsElement,
                    ↪  (IList<ulong>)segment);
                    foreach (var partiallyMatchedSequence in partiallyMatched)
                    {
                        TLink sequenceIndex = (Integer<TLink>)partiallyMatchedSequence;
                        duplicates.Add(sequenceIndex);
                    }
                }
                duplicates.Sort();
                return duplicates;
            }

            private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
            {
                if (!(_links is ILinks<ulong> ulongLinks))
                {
                    return;
                }
                var duplicatesKey = duplicatesItem.Key;
                var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
                Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)})");
                var duplicatesList = duplicatesItem.Value;
                for (int i = 0; i < duplicatesList.Count; i++)
                {
                    ulong sequenceIndex = (Integer<TLink>)duplicatesList[i];
                    var formatedSequenceStructure = ulongLinks.FormatStructure(sequenceIndex, x =>
                    ↪  Point<ulong>.IsPartialPoint(x), (sb, link) => _ =
                    ↪  UnicodeMap.IsCharLink(link.Index) ?
                    ↪  sb.Append(UnicodeMap.FromLinkToChar(link.Index)) : sb.Append(link.Index));
                    Console.WriteLine(formatedSequenceStructure);
                    var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
                    ↪  ulongLinks);
                    Console.WriteLine(sequenceString);
                }
                Console.WriteLine();
            }
        }
    }
```

./Platform.Data.Doublets/Sequences/Frequencies/Cache/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;

```

```
10          public FrequenciesCacheBasedLinkFrequencyIncrementer(LinkFrequenciesCache<TLink> cache)
            ↪ => _cache = cache;
11
12          /// <remarks>Sequence itseft is not changed, only frequency of its doublets is
            ↪ incremented.</remarks>
13          public IList<TLink> Increment(IList<TLink> sequence)
14          {
15              _cache.IncrementFrequencies(sequence);
16              return sequence;
17          }
18      }
19  }
```

./Platform.Data.Doublets/Sequences/Frequencies/Cache/FrequenciesCacheBasedLinkToItsFrequencyNumberConve

```
1   using Platform.Interfaces;
2
3   namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
4   {
5       public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLink> :
            ↪ IConverter<Doublet<TLink>, TLink>
6       {
7           private readonly LinkFrequenciesCache<TLink> _cache;
8           public
                ↪ FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink>
                ↪ cache) => _cache = cache;
9           public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref source).Frequency;
10      }
11  }
```

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

```
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using Platform.Interfaces;
5   using Platform.Numbers;
6
7   namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
8   {
9       /// <remarks>
10      /// Can be used to operate with many CompressingConverters (to keep global frequencies data
            ↪ between them).
11      /// TODO: Extract interface to implement frequencies storage inside Links storage
12      /// </remarks>
13      public class LinkFrequenciesCache<TLink> : LinksOperatorBase<TLink>
14      {
15          private static readonly EqualityComparer<TLink> _equalityComparer =
                ↪ EqualityComparer<TLink>.Default;
16          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
17
18          private readonly Dictionary<Doublet<TLink>, LinkFrequency<TLink>> _doubletsCache;
19          private readonly ICounter<TLink, TLink> _frequencyCounter;
20
21          public LinkFrequenciesCache(ILinks<TLink> links, ICounter<TLink, TLink> frequencyCounter)
22              : base(links)
23          {
24              _doubletsCache = new Dictionary<Doublet<TLink>, LinkFrequency<TLink>>(4096,
                    ↪ DoubletComparer<TLink>.Default);
25              _frequencyCounter = frequencyCounter;
26          }
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
30          {
31              var doublet = new Doublet<TLink>(source, target);
32              return GetFrequency(ref doublet);
33          }
34
35          [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 = Arithmetic.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(Arithmetic.Subtract(frequency, count),
                        ↪  Integer<TLink>.One) > 0))
103                         || ((_comparer.Compare(count, frequency) > 0) &&
                            ↪  (_comparer.Compare(Arithmetic.Subtract(count, frequency),
                            ↪  Integer<TLink>.One) > 0)))
104                     {
105                         throw new InvalidOperationException("Frequencies validation failed.");
106                     }
107                 }
108                 //else
109                 //{
110                 //    if (value.Frequency > 0)
111                 //    {
112                 //        var frequency = value.Frequency;
113                 //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
114                 //        var count = _countLinkFrequency(linkIndex);
115
116                 //        if ((frequency > count && frequency - count > 1) || (count > frequency
                    ↪  && count - frequency > 1))
117                 //            throw new Exception("Frequencies validation failed.");
118                 //    }
```

```
                    //}
                }
            }
        }
    }
```

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

```
using System.Runtime.CompilerServices;
using Platform.Numbers;

namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
{
    public class LinkFrequency<TLink>
    {
        public TLink Frequency { get; set; }
        public TLink Link { get; set; }

        public LinkFrequency(TLink frequency, TLink link)
        {
            Frequency = frequency;
            Link = link;
        }

        public LinkFrequency() { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void IncrementFrequency() => Frequency = Arithmetic<TLink>.Increment(Frequency);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void DecrementFrequency() => Frequency = Arithmetic<TLink>.Decrement(Frequency);

        public override string ToString() => $"F: {Frequency}, L: {Link}";
    }
}
```

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

```
using Platform.Interfaces;

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

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

        public override TLink Count()
        {
            if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
            {
                return default;
            }
            return base.Count();
        }
    }
}
```

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

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    public class SequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        protected readonly ILinks<TLink> _links;
        protected readonly TLink _sequenceLink;
        protected readonly TLink _symbol;
        protected TLink _total;

```

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

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

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

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

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

```
        }
    }
}
```

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

```
using Platform.Interfaces;

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

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

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    public class TotalSequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        protected readonly ILinks<TLink> _links;
        protected readonly TLink _symbol;
        protected readonly HashSet<TLink> _visits;
        protected TLink _total;

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

        public TLink Count()
        {
            if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
            {
                return _total;
            }
            CountCore(_symbol);
            return _total;
        }

        private void CountCore(TLink link)
        {
            var any = _links.Constants.Any;
            if (_equalityComparer.Equals(_links.Count(any, link), default))
            {
                CountSequenceSymbolFrequency(link);
            }
            else
            {
                _links.Each(EachElementHandler, any, link);
            }
        }

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

        private TLink EachElementHandler(IList<TLink> doublet)
        {
            var constants = _links.Constants;
            var doubletIndex = doublet[constants.IndexPart];
            if (_visits.Add(doubletIndex))
            {
```

```
                CountCore(doubletIndex);
            }
            return constants.Continue;
        }
    }
}
```

./Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs

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

namespace Platform.Data.Doublets.Sequences.HeightProviders
{
    public class CachedSequenceHeightProvider<TLink> : LinksOperatorBase<TLink>, ISequenceHeightProvider<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer = EqualityComparer<TLink>.Default;

        private readonly TLink _heightPropertyMarker;
        private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
        private readonly IConverter<TLink> _addressToUnaryNumberConverter;
        private readonly IConverter<TLink> _unaryNumberToAddressConverter;
        private readonly IPropertiesOperator<TLink, TLink, TLink> _propertyOperator;

        public CachedSequenceHeightProvider(
            ILinks<TLink> links,
            ISequenceHeightProvider<TLink> baseHeightProvider,
            IConverter<TLink> addressToUnaryNumberConverter,
            IConverter<TLink> unaryNumberToAddressConverter,
            TLink heightPropertyMarker,
            IPropertiesOperator<TLink, TLink, TLink> propertyOperator)
            : base(links)
        {
            _heightPropertyMarker = heightPropertyMarker;
            _baseHeightProvider = baseHeightProvider;
            _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
            _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
            _propertyOperator = propertyOperator;
        }

        public TLink Get(TLink sequence)
        {
            TLink height;
            var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
            if (_equalityComparer.Equals(heightValue, default))
            {
                height = _baseHeightProvider.Get(sequence);
                heightValue = _addressToUnaryNumberConverter.Convert(height);
                _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
            }
            else
            {
                height = _unaryNumberToAddressConverter.Convert(heightValue);
            }
            return height;
        }
    }
}
```

./Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs

```
using Platform.Interfaces;
using Platform.Numbers;

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

        public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriterionMatcher<TLink> elementMatcher) : base(links) => _elementMatcher = elementMatcher;

        public TLink Get(TLink sequence)
        {
            var height = default(TLink);
            var pairOrElement = sequence;
            while (!_elementMatcher.IsMatched(pairOrElement))
            {
```

```
                pairOrElement = Links.GetTarget(pairOrElement);
                height = Arithmetic.Increment(height);
            }
            return height;
        }
    }
}
```

./Platform.Data.Doublets/Sequences/HeightProviders/ISequenceHeightProvider.cs

```
using Platform.Interfaces;

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

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

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

namespace Platform.Data.Doublets.Sequences
{
    /// <summary>
    /// Представляет коллекцию последовательностей связей.
    /// </summary>
    /// <remarks>
    /// Обязательно реализовать атомарность каждого публичного метода.
    ///
    /// TODO:
    ///
    /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
    /// через естественную группировку по unicode типам, все whitespace вместе, все символы вместе, все числа вместе и т.п.
    /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину графа)
    ///
    /// x*y - найти все связи между, в последовательностях любой формы, если не стоит ограничитель на то, что является последовательностью, а что нет,
    /// то находятся любые структуры связей, которые содержат эти элементы именно в таком порядке.
    ///
    /// Рост последовательности слева и справа.
    /// Поиск со звёздочкой.
    /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
    /// так же проблема может быть решена при реализации дистанционных триггеров.
    /// Нужны ли уникальные указатели вообще?
    /// Что если обращение к информации будет происходить через содержимое всегда?
    ///
    /// Писать тесты.
    ///
    ///
    /// Можно убрать зависимость от конкретной реализации Links,
    /// на зависимость от абстрактного элемента, который может быть представлен несколькими способами.
    ///
    /// Можно ли как-то сделать один общий интерфейс
    ///
    ///
    /// Блокчейн и/или гит для распределённой записи транзакций.
    ///
    /// </remarks>
    public partial class Sequences : ISequences<ulong> // IList<string>, IList<ulong[]> (после завершения реализации Sequences)
    {
        private static readonly LinksCombinedConstants<bool, ulong, long> _constants = Default<LinksCombinedConstants<bool, ulong, long>>.Instance;

```

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

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

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

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

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

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

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

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

        #region Count

        public ulong Count(params ulong[] sequence)
        {
            if (sequence.Length == 0)
            {
                return Links.Count(_constants.Any, Options.SequenceMarkerLink, _constants.Any);
            }
            if (sequence.Length == 1) // Первая связь это адрес
            {
                if (sequence[0] == _constants.Null)
                {
                    return 0;
                }
                if (sequence[0] == _constants.Any)
                {
                    return Count();
                }
```

```
            if (Options.UseSequenceMarker)
            {
                return Links.Count(_constants.Any, Options.SequenceMarkerLink, sequence[0]);
            }
            return Links.Exists(sequence[0]) ? 1UL : 0;
        }
        throw new NotImplementedException();
    }

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

    #endregion

    #region Create

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

    private ulong CreateCore(params ulong[] sequence)
    {
        if (Options.UseIndex)
        {
            Options.Indexer.Index(sequence);
        }
        var sequenceRoot = default(ulong);
        if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
        {
            var matches = Each(sequence);
            if (matches.Count > 0)
            {
                sequenceRoot = matches[0];
            }
        }
        else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
        {
            return CompactCore(sequence);
        }
        if (sequenceRoot == default)
        {
            sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
        }
        if (Options.UseSequenceMarker)
        {
```

```
                Links.Unsync.CreateAndUpdate(Options.SequenceMarkerLink, sequenceRoot);
            }
            return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
        }

        #endregion

        #region Each

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

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

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

        private bool PartialStepRight(Func<ulong, bool> handler, ulong left, ulong right)
        {
```

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

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

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

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

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

        #endregion

        #region Update

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

```
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
```

```
        #region Delete

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

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

        #endregion

        #region Compactification

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

```
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)
```

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

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

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

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

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

        #endregion
    }
}
```

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

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

```

```
14  namespace Platform.Data.Doublets.Sequences
15  {
16      partial class Sequences
17      {
18          #region Create All Variants (Not Practical)
19
20          /// <remarks>
21          /// Number of links that is needed to generate all variants for
22          /// sequence of length N corresponds to https://oeis.org/A014143/list sequence.
23          /// </remarks>
24          public ulong[] CreateAllVariants2(ulong[] sequence)
25          {
26              return Sync.ExecuteWriteOperation(() =>
27              {
28                  if (sequence.IsNullOrEmpty())
29                  {
30                      return new ulong[0];
31                  }
32                  Links.EnsureEachLinkExists(sequence);
33                  if (sequence.Length == 1)
34                  {
35                      return sequence;
36                  }
37                  return CreateAllVariants2Core(sequence, 0, sequence.Length - 1);
38              });
39          }
40
41          private ulong[] CreateAllVariants2Core(ulong[] sequence, long startAt, long stopAt)
42          {
43  #if DEBUG
44              if ((stopAt - startAt) < 0)
45              {
46                  throw new ArgumentOutOfRangeException(nameof(startAt), "startAt должен быть
                        ↪ меньше или равен stopAt");
47              }
48  #endif
49              if ((stopAt - startAt) == 0)
50              {
51                  return new[] { sequence[startAt] };
52              }
53              if ((stopAt - startAt) == 1)
54              {
55                  return new[] { Links.Unsync.CreateAndUpdate(sequence[startAt], sequence[stopAt])
                        ↪ };
56              }
57              var variants = new ulong[(ulong)Numbers.Math.Catalan(stopAt - startAt)];
58              var last = 0;
59              for (var splitter = startAt; splitter < stopAt; splitter++)
60              {
61                  var left = CreateAllVariants2Core(sequence, startAt, splitter);
62                  var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
63                  for (var i = 0; i < left.Length; i++)
64                  {
65                      for (var j = 0; j < right.Length; j++)
66                      {
67                          var variant = Links.Unsync.CreateAndUpdate(left[i], right[j]);
68                          if (variant == _constants.Null)
69                          {
70                              throw new NotImplementedException("Creation cancellation is not
                                    ↪ implemented.");
71                          }
72                          variants[last++] = variant;
73                      }
74                  }
75              }
76              return variants;
77          }
78
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)
```

```
            {
                return new List<ulong> { sequence[0] };
            }
            var results = new List<ulong>((int)Numbers.Math.Catalan(sequence.Length));
            return CreateAllVariants1Core(sequence, results);
        });
    }

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

    #endregion

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

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

```
                ulong[] innerSequence = null;
                Links.Unsync.Each(left, right, doublet =>
                {
                    if (innerSequence == null)
                    {
                        innerSequence = new ulong[innerSequenceLength];
                        for (var isi = 0; isi < linkIndex; isi++)
                        {
                            innerSequence[isi] = sequence[isi];
                        }
                        for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
                        {
                            innerSequence[isi] = sequence[isi + 1];
                        }
                    }
                    innerSequence[linkIndex] = doublet;
                    Each1(handler, innerSequence);
                    return _constants.Continue;
                });
            }
        }
    }

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

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

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

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

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

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

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

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

```

```
323         private void StepLeft(Action<ulong> handler, ulong left, ulong right)
324         {
325             Links.Unsync.Each(_constants.Any, right, leftStep =>
326             {
327                 TryStepLeftUp(handler, left, leftStep);
328                 return true;
329             });
330         }
331 
332         private void TryStepLeftUp(Action<ulong> handler, ulong left, ulong stepFrom)
333         {
334             var upStep = stepFrom;
335             var firstTarget = Links.Unsync.GetSource(upStep);
336             while (firstTarget != left && firstTarget != upStep)
337             {
338                 upStep = firstTarget;
339                 firstTarget = Links.Unsync.GetTarget(upStep);
340             }
341             if (firstTarget == left)
342             {
343                 handler(stepFrom);
344             }
345         }
346 
347         private bool StartsWith(ulong sequence, ulong link)
348         {
349             var upStep = sequence;
350             var firstSource = Links.Unsync.GetSource(upStep);
351             while (firstSource != link && firstSource != upStep)
352             {
353                 upStep = firstSource;
354                 firstSource = Links.Unsync.GetSource(upStep);
355             }
356             return firstSource == link;
357         }
358 
359         private bool EndsWith(ulong sequence, ulong link)
360         {
361             var upStep = sequence;
362             var lastTarget = Links.Unsync.GetTarget(upStep);
363             while (lastTarget != link && lastTarget != upStep)
364             {
365                 upStep = lastTarget;
366                 lastTarget = Links.Unsync.GetTarget(upStep);
367             }
368             return lastTarget == link;
369         }
370 
371         public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
372         {
373             return Sync.ExecuteReadOperation(() =>
374             {
375                 var results = new List<ulong>();
376                 if (sequence.Length > 0)
377                 {
378                     Links.EnsureEachLinkExists(sequence);
379                     var firstElement = sequence[0];
380                     if (sequence.Length == 1)
381                     {
382                         results.Add(firstElement);
383                         return results;
384                     }
385                     if (sequence.Length == 2)
386                     {
387                         var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
388                         if (doublet != _constants.Null)
389                         {
390                             results.Add(doublet);
391                         }
392                         return results;
393                     }
394                     var linksInSequence = new HashSet<ulong>(sequence);
395                     void handler(ulong result)
396                     {
397                         var filterPosition = 0;
398                         StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
                            ↪  Links.Unsync.GetTarget,
399                             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]);
                }
```

```
475                 }
476                 return results;
477             });
478         }
479 
480         public const int MaxSequenceFormatSize = 200;
481 
482         public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
            => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
483 
484         public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
            elementToString, bool insertComma, params LinkIndex[] knownElements) =>
            Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
            elementToString, insertComma, knownElements));
485 
486         private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
            Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
            LinkIndex[] knownElements)
487         {
488             var linksInSequence = new HashSet<ulong>(knownElements);
489             //var entered = new HashSet<ulong>();
490             var sb = new StringBuilder();
491             sb.Append('{');
492             if (links.Exists(sequenceLink))
493             {
494                 StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
495                     x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
                        entered.AddAndReturnVoid, x => { }, entered.DoNotContains
496                     {
497                         if (insertComma && sb.Length > 1)
498                         {
499                             sb.Append(',');
500                         }
501                         //if (entered.Contains(element))
502                         //{
503                         //    sb.Append('{');
504                         //    elementToString(sb, element);
505                         //    sb.Append('}');
506                         //}
507                         //else
508                         elementToString(sb, element);
509                         if (sb.Length < MaxSequenceFormatSize)
510                         {
511                             return true;
512                         }
513                         sb.Append(insertComma ? ", ..." : "...");
514                         return false;
515                     });
516             }
517             sb.Append('}');
518             return sb.ToString();
519         }
520 
521         public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
            knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
            knownElements);
522 
523         public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
            LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
            Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
            sequenceLink, elementToString, insertComma, knownElements));
524 
525         private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
            Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
            LinkIndex[] knownElements)
526         {
527             var linksInSequence = new HashSet<ulong>(knownElements);
528             var entered = new HashSet<ulong>();
529             var sb = new StringBuilder();
530             sb.Append('{');
531             if (links.Exists(sequenceLink))
532             {
533                 StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
534                     x => linksInSequence.Contains(x) || links.IsFullPoint(x),
                        entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
535                     {
536                         if (insertComma && sb.Length > 1)
537                         {
```

```
                        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>();
            });
```

```
615         }
616
617         public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
618         {
619             return Sync.ExecuteReadOperation(() =>
620             {
621                 if (sequence.Length > 0)
622                 {
623                     Links.EnsureEachLinkExists(sequence);
624                     var results = new HashSet<ulong>();
625                     for (var i = 0; i < sequence.Length; i++)
626                     {
627                         AllUsagesCore(sequence[i], results);
628                     }
629                     var filteredResults = new HashSet<ulong>();
630                     var matcher = new Matcher(this, sequence, filteredResults, null);
631                     matcher.AddAllPartialMatchedToResults(results);
632                     return filteredResults;
633                 }
634                 return new HashSet<ulong>();
635             });
636         }
637
638         public bool GetAllPartiallyMatchingSequences2(Func<ulong, bool> handler, params ulong[]
    ↪  sequence)
639         {
640             return Sync.ExecuteReadOperation(() =>
641             {
642                 if (sequence.Length > 0)
643                 {
644                     Links.EnsureEachLinkExists(sequence);
645
646                     var results = new HashSet<ulong>();
647                     var filteredResults = new HashSet<ulong>();
648                     var matcher = new Matcher(this, sequence, filteredResults, handler);
649                     for (var i = 0; i < sequence.Length; i++)
650                     {
651                         if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
652                         {
653                             return false;
654                         }
655                     }
656                     return true;
657                 }
658                 return true;
659             });
660         }
661
662         //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
663         //{
664         //    return Sync.ExecuteReadOperation(() =>
665         //    {
666         //        if (sequence.Length > 0)
667         //        {
668         //            _links.EnsureEachLinkIsAnyOrExists(sequence);
669
670         //            var firstResults = new HashSet<ulong>();
671         //            var lastResults = new HashSet<ulong>();
672
673         //            var first = sequence.First(x => x != LinksConstants.Any);
674         //            var last = sequence.Last(x => x != LinksConstants.Any);
675
676         //            AllUsagesCore(first, firstResults);
677         //            AllUsagesCore(last, lastResults);
678
679         //            firstResults.IntersectWith(lastResults);
680
681         //            //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;
757                 }
758                 return new HashSet<ulong>();
759             });
760         }
761
762         // Does not work
763         public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
            ↪ params ulong[] sequence)
764         {
765             var visited = new HashSet<ulong>();
766             var results = new HashSet<ulong>();
```

```
767             var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
                ↪  true; }, readAsElements);
768             var last = sequence.Length - 1;
769             for (var i = 0; i < last; i++)
770             {
771                 PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
772             }
773             return results;
774         }
775 
776         public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
777         {
778             return Sync.ExecuteReadOperation(() =>
779             {
780                 if (sequence.Length > 0)
781                 {
782                     Links.EnsureEachLinkExists(sequence);
783                     //var firstElement = sequence[0];
784                     //if (sequence.Length == 1)
785                     //{
786                     //    //results.Add(firstElement);
787                     //    return results;
788                     //}
789                     //if (sequence.Length == 2)
790                     //{
791                     //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
792                     //    //if (doublet != Doublets.Links.Null)
793                     //    //    results.Add(doublet);
794                     //    return results;
795                     //}
796                     //var lastElement = sequence[sequence.Length - 1];
797                     //Func<ulong, bool> handler = x =>
798                     //{
799                     //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
                        ↪  results.Add(x);
800                     //    return true;
801                     //};
802                     //if (sequence.Length >= 2)
803                     //    StepRight(handler, sequence[0], sequence[1]);
804                     //var last = sequence.Length - 2;
805                     //for (var i = 1; i < last; i++)
806                     //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
807                     //if (sequence.Length >= 3)
808                     //    StepLeft(handler, sequence[sequence.Length - 2],
                        ↪  sequence[sequence.Length - 1]);
809                     //////if (sequence.Length == 1)
810                     //////{
811                     //////    throw new NotImplementedException(); // all sequences, containing
                        ↪  this element?
812                     //////}
813                     //////if (sequence.Length == 2)
814                     //////{
815                     //////    var results = new List<ulong>();
816                     //////    PartialStepRight(results.Add, sequence[0], sequence[1]);
817                     //////    return results;
818                     //////}
819                     //////var matches = new List<List<ulong>>();
820                     //////var last = sequence.Length - 1;
821                     //////for (var i = 0; i < last; i++)
822                     //////{
823                     //////    var results = new List<ulong>();
824                     //////    //StepRight(results.Add, sequence[i], sequence[i + 1]);
825                     //////    PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
826                     //////    if (results.Count > 0)
827                     //////        matches.Add(results);
828                     //////    else
829                     //////        return results;
830                     //////    if (matches.Count == 2)
831                     //////    {
832                     //////        var merged = new List<ulong>();
833                     //////        for (var j = 0; j < matches[0].Count; j++)
834                     //////            for (var k = 0; k < matches[1].Count; k++)
835                     //////                CloseInnerConnections(merged.Add, matches[0][j],
                        ↪  matches[1][k]);
836                     //////        if (merged.Count > 0)
837                     //////            matches = new List<List<ulong>> { merged };
838                     //////        else
```

```
839                     //////            return new List<ulong>();
840                     //////    }
841                     //////}
842                     //////if (matches.Count > 0)
843                     //////{
844                     //////    var usages = new HashSet<ulong>();
845                     //////    for (int i = 0; i < sequence.Length; i++)
846                     //////    {
847                     //////        AllUsagesCore(sequence[i], usages);
848                     //////    }
849                     //////    //for (int i = 0; i < matches[0].Count; i++)
850                     //////    //    AllUsagesCore(matches[0][i], usages);
851                     //////    //usages.UnionWith(matches[0]);
852                     //////    return usages.ToList();
853                     //////}
854                     var firstLinkUsages = new HashSet<ulong>();
855                     AllUsagesCore(sequence[0], firstLinkUsages);
856                     firstLinkUsages.Add(sequence[0]);
857                     //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
                        ↪  sequence[0] }; // or all sequences, containing this element?
858                     //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
                        ↪  1).ToList();
859                     var results = new HashSet<ulong>();
860                     foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
                        ↪  firstLinkUsages, 1))
861                     {
862                         AllUsagesCore(match, results);
863                     }
864                     return results.ToList();
865                 }
866                 return new List<ulong>();
867             });
868         }
869 
870         /// <remarks>
871         /// TODO: Может потробоваться ограничение на уровень глубины рекурсии
872         /// </remarks>
873         public HashSet<ulong> AllUsages(ulong link)
874         {
875             return Sync.ExecuteReadOperation(() =>
876             {
877                 var usages = new HashSet<ulong>();
878                 AllUsagesCore(link, usages);
879                 return usages;
880             });
881         }
882 
883         // При сборе всех использований (последовательностей) можно сохранять обратный путь к
                ↪  той связи с которой начинался поиск (STTTSSSTT),
884         // причём достаточно одного бита для хранения перехода влево или вправо
885         private void AllUsagesCore(ulong link, HashSet<ulong> usages)
886         {
887             bool handler(ulong doublet)
888             {
889                 if (usages.Add(doublet))
890                 {
891                     AllUsagesCore(doublet, usages);
892                 }
893                 return true;
894             }
895             Links.Unsync.Each(link, _constants.Any, handler);
896             Links.Unsync.Each(_constants.Any, link, handler);
897         }
898 
899         public HashSet<ulong> AllBottomUsages(ulong link)
900         {
901             return Sync.ExecuteReadOperation(() =>
902             {
903                 var visits = new HashSet<ulong>();
904                 var usages = new HashSet<ulong>();
905                 AllBottomUsagesCore(link, visits, usages);
906                 return usages;
907             });
908         }
909 
910         private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
                ↪  usages)
911         {
```

```
912             bool handler(ulong doublet)
913             {
914                 if (visits.Add(doublet))
915                 {
916                     AllBottomUsagesCore(doublet, visits, usages);
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                    {
1051                        _totals[parent]++;
1052                    }
1053                }
1054                var stack = new Stack();
1055                var element = link;
1056                if (isElement(element))
1057                {
1058                    visitLeaf(element);
1059                }
1060                else
1061                {
1062                    while (true)
1063                    {
```

```
                    if (isElement(element))
                    {
                        if (stack.Count == 0)
                        {
                            break;
                        }
                        element = stack.Pop();
                        var source = getSource(element);
                        var target = getTarget(element);
                        // Обработка элемента
                        if (isElement(target))
                        {
                            visitLeaf(target);
                        }
                        if (isElement(source))
                        {
                            visitLeaf(source);
                        }
                        element = source;
                    }
                    else
                    {
                        stack.Push(element);
                        visitNode(element);
                        element = getTarget(element);
                    }
                }
            }
            _totals[link]++;
            return true;
        }
    }

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

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

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

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

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

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

    private class AllUsagesCollector2
```

```
        {
            private readonly ILinks<ulong> _links;
            private readonly BitString _usages;

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

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

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

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

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

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

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

```

```
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                    {
1278                        if (patternSequence[i] != _constants.Any && patternSequence[i] !=
                            ↪ ZeroOrMany)
1279                        {
1280                            uniqueSequenceElements.Add(patternSequence[i]);
1281                        }
1282                    }
1283                    var results = new HashSet<ulong>();
1284                    foreach (var uniqueSequenceElement in uniqueSequenceElements)
1285                    {
1286                        AllUsagesCore(uniqueSequenceElement, results);
1287                    }
1288                    var filteredResults = new HashSet<ulong>();
```

```
                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;
        });
```

```
1366        }
1367
1368        public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
1369        {
1370            return Sync.ExecuteReadOperation(() =>
1371            {
1372                var results = new BitString((long)Links.Unsync.Count() + 1); // new
                    ↪  BitArray((int)_links.Total + 1);
1373                if (linksToConnect.Length > 0)
1374                {
1375                    Links.EnsureEachLinkExists(linksToConnect);
1376                    var collector1 = new AllUsagesCollector2(Links.Unsync, results);
1377                    collector1.Collect(linksToConnect[0]);
1378                    for (var i = 1; i < linksToConnect.Length; i++)
1379                    {
1380                        var next = new BitString((long)Links.Unsync.Count() + 1); //new
                            ↪  BitArray((int)_links.Total + 1);
1381                        var collector = new AllUsagesCollector2(Links.Unsync, next);
1382                        collector.Collect(linksToConnect[i]);
1383                        results = results.And(next);
1384                    }
1385                }
1386                return results.GetSetUInt64Indices();
1387            });
1388        }
1389
1390        private static ulong[] Simplify(ulong[] sequence)
1391        {
1392            // Считаем новый размер последовательности
1393            long newLength = 0;
1394            var zeroOrManyStepped = false;
1395            for (var i = 0; i < sequence.Length; i++)
1396            {
1397                if (sequence[i] == ZeroOrMany)
1398                {
1399                    if (zeroOrManyStepped)
1400                    {
1401                        continue;
1402                    }
1403                    zeroOrManyStepped = true;
1404                }
1405                else
1406                {
1407                    //if (zeroOrManyStepped) Is it efficient?
1408                    zeroOrManyStepped = false;
1409                }
1410                newLength++;
1411            }
1412            // Строим новую последовательность
1413            zeroOrManyStepped = false;
1414            var newSequence = new ulong[newLength];
1415            long j = 0;
1416            for (var i = 0; i < sequence.Length; i++)
1417            {
1418                //var current = zeroOrManyStepped;
1419                //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
1420                //if (current && zeroOrManyStepped)
1421                //    continue;
1422                //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1423                //if (zeroOrManyStepped && newZeroOrManyStepped)
1424                //    continue;
1425                //zeroOrManyStepped = newZeroOrManyStepped;
1426                if (sequence[i] == ZeroOrMany)
1427                {
1428                    if (zeroOrManyStepped)
1429                    {
1430                        continue;
1431                    }
1432                    zeroOrManyStepped = true;
1433                }
1434                else
1435                {
1436                    //if (zeroOrManyStepped) Is it efficient?
1437                    zeroOrManyStepped = false;
1438                }
1439                newSequence[j++] = sequence[i];
1440            }
1441            return newSequence;
1442        }
```

```
1443
1444            public static void TestSimplify()
1445            {
1446                var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany,
                    ↪  ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
1447                var simplifiedSequence = Simplify(sequence);
1448            }
1449
1450            public List<ulong> GetSimilarSequences() => new List<ulong>();
1451
1452            public void Prediction()
1453            {
1454                //_links
1455                //sequences
1456            }
1457
1458            #region From Triplets
1459
1460            //public static void DeleteSequence(Link sequence)
1461            //{
1462            //}
1463
1464            public List<ulong> CollectMatchingSequences(ulong[] links)
1465            {
1466                if (links.Length == 1)
1467                {
1468                    throw new Exception("Подпоследовательности с одним элементом не
                        ↪  поддерживаются.");
1469                }
1470                var leftBound = 0;
1471                var rightBound = links.Length - 1;
1472                var left = links[leftBound++];
1473                var right = links[rightBound--];
1474                var results = new List<ulong>();
1475                CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
1476                return results;
1477            }
1478
1479            private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
                ↪  middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1480            {
1481                var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1482                var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1483                if (leftLinkTotalReferers <= rightLinkTotalReferers)
1484                {
1485                    var nextLeftLink = middleLinks[leftBound];
1486                    var elements = GetRightElements(leftLink, nextLeftLink);
1487                    if (leftBound <= rightBound)
1488                    {
1489                        for (var i = elements.Length - 1; i >= 0; i--)
1490                        {
1491                            var element = elements[i];
1492                            if (element != 0)
1493                            {
1494                                CollectMatchingSequences(element, leftBound + 1, middleLinks,
                                    ↪  rightLink, rightBound, ref results);
1495                            }
1496                        }
1497                    }
1498                    else
1499                    {
1500                        for (var i = elements.Length - 1; i >= 0; i--)
1501                        {
1502                            var element = elements[i];
1503                            if (element != 0)
1504                            {
1505                                results.Add(element);
1506                            }
1507                        }
1508                    }
1509                }
1510                else
1511                {
1512                    var nextRightLink = middleLinks[rightBound];
1513                    var elements = GetLeftElements(rightLink, nextRightLink);
1514                    if (leftBound <= rightBound)
1515                    {
1516                        for (var i = elements.Length - 1; i >= 0; i--)
```

```
                {
                    var element = elements[i];
                    if (element != 0)
                    {
                        CollectMatchingSequences(leftLink, leftBound, middleLinks,
                        ↪  elements[i], rightBound - 1, ref results);
                    }
                }
            }
            else
            {
                for (var i = elements.Length - 1; i >= 0; i--)
                {
                    var element = elements[i];
                    if (element != 0)
                    {
                        results.Add(element);
                    }
                }
            }
        }
    }

    public ulong[] GetRightElements(ulong startLink, ulong rightLink)
    {
        var result = new ulong[5];
        TryStepRight(startLink, rightLink, result, 0);
        Links.Each(_constants.Any, startLink, couple =>
        {
            if (couple != startLink)
            {
                if (TryStepRight(couple, rightLink, result, 2))
                {
                    return false;
                }
            }
            return true;
        });
        if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
        {
            result[4] = startLink;
        }
        return result;
    }

    public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
    {
        var added = 0;
        Links.Each(startLink, _constants.Any, couple =>
        {
            if (couple != startLink)
            {
                var coupleTarget = Links.GetTarget(couple);
                if (coupleTarget == rightLink)
                {
                    result[offset] = couple;
                    if (++added == 2)
                    {
                        return false;
                    }
                }
                else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker
                ↪  == Net.And &&
                {
                    result[offset + 1] = couple;
                    if (++added == 2)
                    {
                        return false;
                    }
                }
            }
            return true;
        });
        return added > 0;
    }

    public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
    {
```

```
            var result = new ulong[5];
            TryStepLeft(startLink, leftLink, result, 0);
            Links.Each(startLink, _constants.Any, couple =>
            {
                if (couple != startLink)
                {
                    if (TryStepLeft(couple, leftLink, result, 2))
                    {
                        return false;
                    }
                }
                return true;
            });
            if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
            {
                result[4] = leftLink;
            }
            return result;
        }

        public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
        {
            var added = 0;
            Links.Each(_constants.Any, startLink, couple =>
            {
                if (couple != startLink)
                {
                    var coupleSource = Links.GetSource(couple);
                    if (coupleSource == leftLink)
                    {
                        result[offset] = couple;
                        if (++added == 2)
                        {
                            return false;
                        }
                    }
                    else if (Links.GetTarget(coupleSource) == leftLink) // coupleSource.Linker
                    ↪  == Net.And &&
                    {
                        result[offset + 1] = couple;
                        if (++added == 2)
                        {
                            return false;
                        }
                    }
                }
                return true;
            });
            return added > 0;
        }

        #endregion

        #region Walkers

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

            #region Pattern Match

            enum PatternBlockType
            {
                Undefined,
                Gap,
                Elements
            }

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

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

```
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,
1737                                Start = 1,
1738                                Stop = 1
1739                            };
1740                        }
1741                        else if (_patternSequence[i] == ZeroOrMany)
1742                        {
1743                            pattern.Add(patternBlock);
1744                            patternBlock = new PatternBlock
1745                            {
1746                                Type = PatternBlockType.Gap,
1747                                Start = 0,
1748                                Stop = long.MaxValue
```

```
                    };
                }
                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;
```

```
1827                //    element = 0;
1828                //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
1829                //    {
1830                //        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
1831                //            mininumGap++;
1832                //        else if (_patternSequence[_patternPosition] == ZeroOrMany)
1833                //            maximumGap = long.MaxValue;
1834                //        else
1835                //            break;
1836                //    }
1837
1838                //    if (maximumGap < mininumGap)
1839                //        maximumGap = mininumGap;
1840                //}
1841
1842                private bool PatternMatchCore(LinkIndex element)
1843                {
1844                    if (_patternPosition >= _pattern.Count)
1845                    {
1846                        _patternPosition = -2;
1847                        return false;
1848                    }
1849                    var currentPatternBlock = _pattern[_patternPosition];
1850                    if (currentPatternBlock.Type == PatternBlockType.Gap)
1851                    {
1852                        //var currentMatchingBlockLength = (_sequencePosition -
                            ↪  _lastMatchedBlockPosition);
1853                        if (_sequencePosition < currentPatternBlock.Start)
1854                        {
1855                            _sequencePosition++;
1856                            return true; // Двигаемся дальше
1857                        }
1858                        // Это последний блок
1859                        if (_pattern.Count == _patternPosition + 1)
1860                        {
1861                            _patternPosition++;
1862                            _sequencePosition = 0;
1863                            return false; // Полное соответствие
1864                        }
1865                        else
1866                        {
1867                            if (_sequencePosition > currentPatternBlock.Stop)
1868                            {
1869                                return false; // Соответствие невозможно
1870                            }
1871                            var nextPatternBlock = _pattern[_patternPosition + 1];
1872                            if (_patternSequence[nextPatternBlock.Start] == element)
1873                            {
1874                                if (nextPatternBlock.Start < nextPatternBlock.Stop)
1875                                {
1876                                    _patternPosition++;
1877                                    _sequencePosition = 1;
1878                                }
1879                                else
1880                                {
1881                                    _patternPosition += 2;
1882                                    _sequencePosition = 0;
1883                                }
1884                            }
1885                        }
1886                    }
1887                    else // currentPatternBlock.Type == PatternBlockType.Elements
1888                    {
1889                        var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1890                        if (_patternSequence[patternElementPosition] != element)
1891                        {
1892                            return false; // Соответствие невозможно
1893                        }
1894                        if (patternElementPosition == currentPatternBlock.Stop)
1895                        {
1896                            _patternPosition++;
1897                            _sequencePosition = 0;
1898                        }
1899                        else
1900                        {
1901                            _sequencePosition++;
1902                        }
1903                    }
1904                    return true;
```

```
                //if (_patternSequence[_patternPosition] != element)
                //    return false;
                //else
                //{
                //    _sequencePosition++;
                //    _patternPosition++;
                //    return true;
                //}
                ////////
                //if (_filterPosition == _patternSequence.Length)
                //{
                //    _filterPosition = -2; // Длиннее чем нужно
                //    return false;
                //}
                //if (element != _patternSequence[_filterPosition])
                //{
                //    _filterPosition = -1;
                //    return false; // Начинается иначе
                //}
                //_filterPosition++;
                //if (_filterPosition == (_patternSequence.Length - 1))
                //    return false;
                //if (_filterPosition >= 0)
                //{
                //    if (element == _patternSequence[_filterPosition + 1])
                //        _filterPosition++;
                //    else
                //        return false;
                //}
                //if (_filterPosition < 0)
                //{
                //    if (element == _patternSequence[0])
                //        _filterPosition = 0;
                //}
            }

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

        #endregion
    }
}
```

./Platform.Data.Doublets/Sequences/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;
        }
    }
}
```

./Platform.Data.Doublets/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);
        }
    }
}
```

./Platform.Data.Doublets/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;
        }
    }
}
```

./Platform.Data.Doublets/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; }
        public bool EnforceSingleSequenceVersionOnWriteBasedOnNew { get; set; }

        public MarkedSequenceCreteriaMatcher<TLink> MarkedSequenceMatcher { get; set; }
        public IConverter<IList<TLink>, TLink> LinksToSequenceConverter { get; set; }
        public SequencesIndexer<TLink> Indexer { get; set; }

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

```
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              {
94                  throw new NotSupportedException("To use garbage collection UseSequenceMarker
                    ↪  option must be on.");
95              }
96          }
97      }
98  }
```

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

```
1   using System;
2   using System.Collections.Generic;
```

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

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

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

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

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

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

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

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

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

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

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

        public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
        {
```

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

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

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

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

        public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
        {
            var result = new List<ulong[]>();
            var offset = 0;
            while (offset < sequence.Length)
            {
                var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
                var relativeLength = 1;
                var absoluteLength = offset + relativeLength;
                while (absoluteLength < sequence.Length &&
                       currentCategory ==
                       CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
                {
                    relativeLength++;
                    absoluteLength++;
                }
                // char array to ulong array
                var innerSequence = new ulong[relativeLength];
                var maxLength = offset + relativeLength;
                for (var i = offset; i < maxLength; i++)
                {
                    innerSequence[i - offset] = FromCharToLink(sequence[i]);
                }
                result.Add(innerSequence);
                offset += relativeLength;
            }
            return result;
        }

        public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
        {
            var result = new List<ulong[]>();
            var offset = 0;
            while (offset < array.Length)
            {
                var relativeLength = 1;
                if (array[offset] <= LastCharLink)
                {
                    var currentCategory =
                    CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
```

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

./Platform.Data.Doublets/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;
                }
            }
        }
    }
}
```

./Platform.Data.Doublets/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;
                }
            }
        }
    }
}
```

./Platform.Data.Doublets/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);
    }
}
```

./Platform.Data.Doublets/Stacks/Stack.cs

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

namespace Platform.Data.Doublets.Stacks
{
    public class Stack<TLink> : IStack<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

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

        public bool IsEmpty => _equalityComparer.Equals(Peek(), _stack);

        public Stack(ILinks<TLink> links, TLink stack)
        {
            _links = links;
            _stack = stack;
        }

        private TLink GetStackMarker() => _links.GetSource(_stack);

        private TLink GetTop() => _links.GetTarget(_stack);

        public TLink Peek() => _links.GetTarget(GetTop());

        public TLink Pop()
        {
            var element = Peek();
            if (!_equalityComparer.Equals(element, _stack))
            {
                var top = GetTop();
                var previousTop = _links.GetSource(top);
                _links.Update(_stack, GetStackMarker(), previousTop);
                _links.Delete(top);
            }
            return element;
        }

        public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
        ↪  _links.GetOrCreate(GetTop(), element));
    }
}
```

./Platform.Data.Doublets/Stacks/StackExtensions.cs

```
namespace Platform.Data.Doublets.Stacks
{
    public static class StackExtensions
    {
        public static TLink CreateStack<TLink>(this ILinks<TLink> links, TLink stackMarker)
        {
            var stackPoint = links.CreatePoint();
            var stack = links.Update(stackPoint, stackMarker, stackPoint);
            return stack;
        }

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

./Platform.Data.Doublets/SynchronizedLinks.cs

```
using System;
using System.Collections.Generic;
using Platform.Data.Constants;
```

```
using Platform.Data.Doublets;
using Platform.Threading.Synchronization;

namespace Platform.Data.Doublets
{
    /// <remarks>
    /// TODO: Autogeneration of synchronized wrapper (decorator).
    /// TODO: Try to unfold code of each method using IL generation for performance improvements.
    /// TODO: Or even to unfold multiple layers of implementations.
    /// </remarks>
    public class SynchronizedLinks<T> : ISynchronizedLinks<T>
    {
        public LinksCombinedConstants<T, T, int> Constants { get; }
        public ISynchronization SyncRoot { get; }
        public ILinks<T> Sync { get; }
        public ILinks<T> Unsync { get; }

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

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

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

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

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

./Platform.Data.Doublets/UInt64Link.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using Platform.Exceptions;
using Platform.Ranges;
using Platform.Singletons;
using Platform.Collections.Lists;
using Platform.Data.Constants;

namespace Platform.Data.Doublets
{
    /// <summary>
    /// Структура описывающая уникальную связь.
    /// </summary>
    public struct UInt64Link : IEquatable<UInt64Link>, IReadOnlyList<ulong>, IList<ulong>
    {
        private static readonly LinksCombinedConstants<bool, ulong, int> _constants = Default<LinksCombinedConstants<bool, ulong, int>>.Instance;

        private const int Length = 3;

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

        public static readonly UInt64Link Null = new UInt64Link();

```

```
27          public UInt64Link(params ulong[] values)
28          {
29              Index = values.Length > _constants.IndexPart ? values[_constants.IndexPart] :
                ↪ _constants.Null;
30              Source = values.Length > _constants.SourcePart ? values[_constants.SourcePart] :
                ↪ _constants.Null;
31              Target = values.Length > _constants.TargetPart ? values[_constants.TargetPart] :
                ↪ _constants.Null;
32          }
33
34          public UInt64Link(IList<ulong> values)
35          {
36              Index = values.Count > _constants.IndexPart ? values[_constants.IndexPart] :
                ↪ _constants.Null;
37              Source = values.Count > _constants.SourcePart ? values[_constants.SourcePart] :
                ↪ _constants.Null;
38              Target = values.Count > _constants.TargetPart ? values[_constants.TargetPart] :
                ↪ _constants.Null;
39          }
40
41          public UInt64Link(ulong index, ulong source, ulong target)
42          {
43              Index = index;
44              Source = source;
45              Target = target;
46          }
47
48          public UInt64Link(ulong source, ulong target)
49              : this(_constants.Null, source, target)
50          {
51              Source = source;
52              Target = target;
53          }
54
55          public static UInt64Link Create(ulong source, ulong target) => new UInt64Link(source,
            ↪ target);
56
57          public override int GetHashCode() => (Index, Source, Target).GetHashCode();
58
59          public bool IsNull() => Index == _constants.Null
60                                  && Source == _constants.Null
61                                  && Target == _constants.Null;
62
63          public override bool Equals(object other) => other is UInt64Link &&
            ↪ Equals((UInt64Link)other);
64
65          public bool Equals(UInt64Link other) => Index == other.Index
66                                                  && Source == other.Source
67                                                  && Target == other.Target;
68
69          public static string ToString(ulong index, ulong source, ulong target) => $"({index}:
            ↪ {source}->{target})";
70
71          public static string ToString(ulong source, ulong target) => $"({source}->{target})";
72
73          public static implicit operator ulong[](UInt64Link link) => link.ToArray();
74
75          public static implicit operator UInt64Link(ulong[] linkArray) => new
            ↪ UInt64Link(linkArray);
76
77          public override string ToString() => Index == _constants.Null ? ToString(Source, Target)
            ↪ : ToString(Index, Source, Target);
78
79          #region IList
80
81          public ulong this[int index]
82          {
83              get
84              {
85                  Ensure.Always.ArgumentInRange(index, new Range<int>(0, Length - 1),
                    ↪ nameof(index));
86                  if (index == _constants.IndexPart)
87                  {
88                      return Index;
89                  }
90                  if (index == _constants.SourcePart)
91                  {
92                      return Source;
93                  }
```

```
                if (index == _constants.TargetPart)
                {
                    return Target;
                }
                throw new NotSupportedException(); // Impossible path due to Ensure.ArgumentInRange
            }
            set => throw new NotSupportedException();
        }

        public int Count => Length;

        public bool IsReadOnly => true;

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

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

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

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

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

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

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

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

            return -1;
        }

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

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

        #endregion
    }
}
```

./Platform.Data.Doublets/UInt64LinkExtensions.cs

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

```
}
```

./Platform.Data.Doublets/UInt64LinksExtensions.cs

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

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

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

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

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

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

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

```
71
72          public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
            ↪  Func<UInt64Link, bool> isElement, Action<StringBuilder, UInt64Link> appendElement,
            ↪  bool renderIndex = false, bool renderDebug = false)
73          {
74              var sb = new StringBuilder();
75              var visited = new HashSet<ulong>();
76              links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
                ↪  renderDebug);
77              return sb.ToString();
78          }
79
80          public static void AppendStructure(this ILinks<ulong> links, StringBuilder sb,
            ↪  HashSet<ulong> visited, ulong linkIndex, Func<UInt64Link, bool> isElement,
            ↪  Action<StringBuilder, UInt64Link> appendElement, bool renderIndex = false, bool
            ↪  renderDebug = false)
81          {
82              if (sb == null)
83              {
84                  throw new ArgumentNullException(nameof(sb));
85              }
86              if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
                ↪  Constants.Itself)
87              {
88                  return;
89              }
90              if (links.Exists(linkIndex))
91              {
92                  if (visited.Add(linkIndex))
93                  {
94                      sb.Append('(');
95                      var link = new UInt64Link(links.GetLink(linkIndex));
96                      if (renderIndex)
97                      {
98                          sb.Append(link.Index);
99                          sb.Append(':');
100                     }
101                     if (link.Source == link.Index)
102                     {
103                         sb.Append(link.Index);
104                     }
105                     else
106                     {
107                         var source = new UInt64Link(links.GetLink(link.Source));
108                         if (isElement(source))
109                         {
110                             appendElement(sb, source);
111                         }
112                         else
113                         {
114                             links.AppendStructure(sb, visited, source.Index, isElement,
                                ↪  appendElement, renderIndex);
115                         }
116                     }
117                     sb.Append(' ');
118                     if (link.Target == link.Index)
119                     {
120                         sb.Append(link.Index);
121                     }
122                     else
123                     {
124                         var target = new UInt64Link(links.GetLink(link.Target));
125                         if (isElement(target))
126                         {
127                             appendElement(sb, target);
128                         }
129                         else
130                         {
131                             links.AppendStructure(sb, visited, target.Index, isElement,
                                ↪  appendElement, renderIndex);
132                         }
133                     }
134                     sb.Append(')');
135                 }
136                 else
137                 {
138                     if (renderDebug)
139                     {
```

```
                        sb.Append('*');
                    }
                    sb.Append(linkIndex);
                }
            }
            else
            {
                if (renderDebug)
                {
                    sb.Append('~');
                }
                sb.Append(linkIndex);
            }
        }
    }
}
```

./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs

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

namespace Platform.Data.Doublets
{
    public class UInt64LinksTransactionsLayer : LinksDisposableDecoratorBase<ulong> //-V3073
    {
        /// <remarks>
        /// Альтернативные варианты хранения трансформации (элемента транзакции):
        ///
        /// private enum TransitionType
        /// {
        ///     Creation,
        ///     UpdateOf,
        ///     UpdateTo,
        ///     Deletion
        /// }
        ///
        /// private struct Transition
        /// {
        ///     public ulong TransactionId;
        ///     public UniqueTimestamp Timestamp;
        ///     public TransactionItemType Type;
        ///     public Link Source;
        ///     public Link Linker;
        ///     public Link Target;
        /// }
        ///
        /// Или
        ///
        /// public struct TransitionHeader
        /// {
        ///     public ulong TransactionIdCombined;
        ///     public ulong TimestampCombined;
        ///
        ///     public ulong TransactionId
        ///     {
        ///         get
        ///         {
        ///             return (ulong) mask & TransactionIdCombined;
        ///         }
        ///     }
        ///
        ///     public UniqueTimestamp Timestamp
        ///     {
        ///         get
        ///         {
        ///             return (UniqueTimestamp)mask & TransactionIdCombined;
        ///         }
        ///     }
        ///
```

```
///     public TransactionItemType Type
///     {
///         get
///         {
///             // Использовать по одному биту из TransactionId и Timestamp,
///             // для значения в 2 бита, которое представляет тип операции
///             throw new NotImplementedException();
///         }
///     }
/// }
///
/// private struct Transition
/// {
///     public TransitionHeader Header;
///     public Link Source;
///     public Link Linker;
///     public Link Target;
/// }
///
/// </remarks>
public struct Transition
{
    public static readonly long Size = Structure<Transition>.Size;

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

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

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

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

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

/// <remarks>
/// Другие варианты реализации транзакций (атомарности):
///     1. Разделение хранения значения связи ((Source Target) или (Source Linker
↪  Target)) и индексов.
///     2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
↪  потребуется решить вопрос
///        со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
↪  пересечениями идентификаторов.
///
/// Где хранить промежуточный список транзакций?
///
/// В оперативной памяти:
///  Минусы:
///     1. Может усложнить систему, если она будет функционировать самостоятельно,
///     так как нужно отдельно выделять память под список трансформаций.
///     2. Выделенной оперативной памяти может не хватить, в том случае,
///     если транзакция использует слишком много трансформаций.
///         -> Можно использовать жёсткий диск для слишком длинных транзакций.
///         -> Максимальный размер списка трансформаций можно ограничить / задать
↪  константой.
///     3. При подтверждении транзакции (Commit) все трансформации записываются разом
↪  создавая задержку.
///
/// На жёстком диске:
///  Минусы:
```

```
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 Dispose(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 
217     public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
218 
219     private readonly string _logAddress;
220     private readonly FileStream _log;
221     private readonly Queue<Transition> _transitions;
222     private readonly UniqueTimestampFactory _uniqueTimestampFactory;
223     private Task _transitionsPusher;
224     private Transition _lastCommitedTransition;
225     private ulong _currentTransactionId;
226     private Queue<Transition> _currentTransactionTransitions;
227     private Transaction _currentTransaction;
228     private ulong _lastCommitedTransactionId;
229 
230     public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
231         : base(links)
232     {
233         if (string.IsNullOrWhiteSpace(logAddress))
234         {
235             throw new ArgumentNullException(nameof(logAddress));
236         }
237         // В первой строке файла хранится последняя закоммиченную транзакцию.
238         // При запуске это используется для проверки удачного закрытия файла лога.
239         // In the first line of the file the last committed transaction is stored.
240         // On startup, this is used to check that the log file is successfully closed.
241         var lastCommitedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
242         var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
243         if (!lastCommitedTransition.Equals(lastWrittenTransition))
244         {
245             Dispose();
246             throw new NotSupportedException("Database is damaged, autorecovery is not
                ↪  supported yet.");
247         }
248         if (lastCommitedTransition.Equals(default(Transition)))
249         {
250             FileHelpers.WriteFirst(logAddress, lastCommitedTransition);
251         }
252         _lastCommitedTransition = lastCommitedTransition;
253         // TODO: Think about a better way to calculate or store this value
254         var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
255         _lastCommitedTransactionId = allTransitions.Max(x => x.TransactionId);
256         _uniqueTimestampFactory = new UniqueTimestampFactory();
257         _logAddress = logAddress;
258         _log = FileHelpers.Append(logAddress);
259         _transitions = new Queue<Transition>();
260         _transitionsPusher = new Task(TransitionsPusher);
261         _transitionsPusher.Start();
262     }
263 
264     public IList<ulong> GetLinkValue(ulong link) => Links.GetLink(link);
265 
266     public override ulong Create()
267     {
268         var createdLinkIndex = Links.Create();
269         var createdLink = new UInt64Link(Links.GetLink(createdLinkIndex));
270         CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
            ↪  default, createdLink));
271         return createdLinkIndex;
272     }
273 
274     public override ulong Update(IList<ulong> parts)
275     {
276         var beforeLink = new UInt64Link(Links.GetLink(parts[Constants.IndexPart]));
277         parts[Constants.IndexPart] = Links.Update(parts);
278         var afterLink = new UInt64Link(Links.GetLink(parts[Constants.IndexPart]));
279         CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
            ↪  beforeLink, afterLink));
280         return parts[Constants.IndexPart];
281     }
282 
283     public override void Delete(ulong link)
284     {
```

```
285             var deletedLink = new UInt64Link(Links.GetLink(link));
286             Links.Delete(link);
287             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                    ↪ deletedLink, default));
288         }
289 
290         [MethodImpl(MethodImplOptions.AggressiveInlining)]
291         private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
                ↪ _transitions;
292 
293         private void CommitTransition(Transition transition)
294         {
295             if (_currentTransaction != null)
296             {
297                 Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
298             }
299             var transitions = GetCurrentTransitions();
300             transitions.Enqueue(transition);
301         }
302 
303         private void RevertTransition(Transition transition)
304         {
305             if (transition.After.IsNull()) // Revert Deletion with Creation
306             {
307                 Links.Create();
308             }
309             else if (transition.Before.IsNull()) // Revert Creation with Deletion
310             {
311                 Links.Delete(transition.After.Index);
312             }
313             else // Revert Update
314             {
315                 Links.Update(new[] { transition.After.Index, transition.Before.Source,
                        ↪ transition.Before.Target });
316             }
317         }
318 
319         private void ResetCurrentTransation()
320         {
321             _currentTransactionId = 0;
322             _currentTransactionTransitions = null;
323             _currentTransaction = null;
324         }
325 
326         private void PushTransitions()
327         {
328             if (_log == null || _transitions == null)
329             {
330                 return;
331             }
332             for (var i = 0; i < _transitions.Count; i++)
333             {
334                 var transition = _transitions.Dequeue();
335 
336                 _log.Write(transition);
337                 _lastCommitedTransition = transition;
338             }
339         }
340 
341         private void TransitionsPusher()
342         {
343             while (!IsDisposed && _transitionsPusher != null)
344             {
345                 Thread.Sleep(DefaultPushDelay);
346                 PushTransitions();
347             }
348         }
349 
350         public Transaction BeginTransaction() => new Transaction(this);
351 
352         private void DisposeTransitions()
353         {
354             try
355             {
356                 var pusher = _transitionsPusher;
357                 if (pusher != null)
358                 {
359                     _transitionsPusher = null;
360                     pusher.Wait();
```

```
                }
                if (_transitions != null)
                {
                    PushTransitions();
                }
                _log.DisposeIfPossible();
                FileHelpers.WriteFirst(_logAddress, _lastCommitedTransition);
            }
            catch
            {
            }
        }

        #region DisposalBase

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

        #endregion
    }
}
```

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

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

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

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

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

            ulong x = 10;
            ulong y = 500;

            bool result = false;

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

            var comparer1 = Comparer<ulong>.Default;

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

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

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

```
                }
            });

            var comparer2 = new UInt64Comparer();

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

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

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

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

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

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

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

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

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

            var equalityComparer = EqualityComparer<T>.Default;

            // Create Link
```

```
57              Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));
58
59              var setter = new Setter<T>(constants.Null);
60              links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
61
62              Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
63
64              var linkAddress = links.Create();
65
66              var link = new Link<T>(links.GetLink(linkAddress));
67
68              Assert.True(link.Count == 3);
69              Assert.True(equalityComparer.Equals(link.Index, linkAddress));
70              Assert.True(equalityComparer.Equals(link.Source, constants.Null));
71              Assert.True(equalityComparer.Equals(link.Target, constants.Null));
72
73              Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.One));
74
75              // Get first link
76              setter = new Setter<T>(constants.Null);
77              links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);
78
79              Assert.True(equalityComparer.Equals(setter.Result, linkAddress));
80
81              // Update link to reference itself
82              links.Update(linkAddress, linkAddress, linkAddress);
83
84              link = new Link<T>(links.GetLink(linkAddress));
85
86              Assert.True(equalityComparer.Equals(link.Source, linkAddress));
87              Assert.True(equalityComparer.Equals(link.Target, linkAddress));
88
89              // Update link to reference null (prepare for delete)
90              var updated = links.Update(linkAddress, constants.Null, constants.Null);
91
92              Assert.True(equalityComparer.Equals(updated, linkAddress));
93
94              link = new Link<T>(links.GetLink(linkAddress));
95
96              Assert.True(equalityComparer.Equals(link.Source, constants.Null));
97              Assert.True(equalityComparer.Equals(link.Target, constants.Null));
98
99              // Delete link
100             links.Delete(linkAddress);
101
102             Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));
103
104             setter = new Setter<T>(constants.Null);
105             links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
106
107             Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
108         }
109
110         [Fact]
111         public static void UInt64RawNumbersCRUDTest()
112         {
113             using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪ ResizableDirectMemoryLinks<ulong>>>())
114             {
115                 scope.Use<ILinks<ulong>>().TestRawNumbersCRUDOperations();
116             }
117         }
118
119         [Fact]
120         public static void UInt32RawNumbersCRUDTest()
121         {
122             using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪ ResizableDirectMemoryLinks<uint>>>())
123             {
124                 scope.Use<ILinks<uint>>().TestRawNumbersCRUDOperations();
125             }
126         }
127
128         [Fact]
129         public static void UInt16RawNumbersCRUDTest()
130         {
131             using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪ ResizableDirectMemoryLinks<ushort>>>())
132             {
```

```
            scope.Use<ILinks<ushort>>().TestRawNumbersCRUDOperations();
        }
    }

    [Fact]
    public static void UInt8RawNumbersCRUDTest()
    {
        using (var scope = new Scope<Types<HeapResizableDirectMemory,
        ↪  ResizableDirectMemoryLinks<byte>>>())
        {
            scope.Use<ILinks<byte>>().TestRawNumbersCRUDOperations();
        }
    }

    private static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
    {
        // Constants
        var constants = links.Constants;
        var equalityComparer = EqualityComparer<T>.Default;

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

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

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

        links.Update(linkAddress1, h106E, h108E);

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

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

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

        links.Update(linkAddress2, linkAddress1, h108E);

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

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

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

        links.Update(linkAddress3, linkAddress1, linkAddress2);

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

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

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

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

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

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

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

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

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

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

```

```
            // Delete link
            links.Delete(linkAddress3);

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

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

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

        // TODO: Test layers
    }
}
```

./Platform.Data.Doublets.Tests/EqualityTests.cs

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

namespace Platform.Tests
{
    public static class EqualityTests
    {
        protected class UInt64EqualityComparer : IEqualityComparer<ulong>
        {
            public bool Equals(ulong x, ulong y) => x == y;

            public int GetHashCode(ulong obj) => obj.GetHashCode();
        }

        private static bool Equals1<T>(T x, T y) => Equals(x, y);

        private static bool Equals2<T>(T x, T y) => x.Equals(y);

        private static bool Equals3(ulong x, ulong y) => x == y;

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

            ulong x = 10;
            ulong y = 500;

            bool result = false;

            var ts1 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = Equals1(x, y);
                }
            });

            var ts2 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = Equals2(x, y);
                }
            });

            var ts3 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = Equals3(x, y);
                }
            });

            var equalityComparer1 = EqualityComparer<ulong>.Default;

            var ts4 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = equalityComparer1.Equals(x, y);
                }
```

```
            });

            var equalityComparer2 = new UInt64EqualityComparer();

            var ts5 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = equalityComparer2.Equals(x, y);
                }
            });

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

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

            var comparer = Comparer<ulong>.Default;

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

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

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

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

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

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

        private const long Iterations = 10 * 1024;

        #region Concept

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

            using (var scope = new TempLinksTestScope())
            {
```

```
            var links = scope.Links;

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

                var created = 0;
                var deleted = 0;

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

                    var createPoint = random.NextBoolean();

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

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

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

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

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

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

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

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

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

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

            l2 = links.Update(l2, l1);

            links.Delete(l2);

            Global.Trash = links.Count();

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

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

```
117
118         [Fact]
119         public static void BasicTransactionLogTest()
120         {
121             using (var scope = new TempLinksTestScope(useLog: true))
122             {
123                 var links = scope.Links;
124                 var l1 = links.Create();
125                 var l2 = links.Create();
126
127                 Global.Trash = links.Update(l2, l2, l1, l2);
128
129                 links.Delete(l1);
130
131                 links.Unsync.DisposeIfPossible(); // Close links to access log
132
133                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop
                    ↪ e.TempTransactionLogFilename);
134             }
135         }
136
137         [Fact]
138         public static void TransactionAutoRevertedTest()
139         {
140             // Auto Reverted (Because no commit at transaction)
141             using (var scope = new TempLinksTestScope(useLog: true))
142             {
143                 var links = scope.Links;
144                 var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
145                 using (var transaction = transactionsLayer.BeginTransaction())
146                 {
147                     var l1 = links.Create();
148                     var l2 = links.Create();
149
150                     links.Update(l2, l2, l1, l2);
151                 }
152
153                 Assert.Equal(0UL, links.Count());
154
155                 links.Unsync.DisposeIfPossible();
156
157                 var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s
                    ↪ cope.TempTransactionLogFilename);
158                 Assert.Single(transitions);
159             }
160         }
161
162         [Fact]
163         public static void TransactionUserCodeErrorNoDataSavedTest()
164         {
165             // User Code Error (Autoreverted), no data saved
166             var itself = _constants.Itself;
167
168             TempLinksTestScope lastScope = null;
169             try
170             {
171                 using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
                    ↪ useLog: true))
172                 {
173                     var links = scope.Links;
174                     var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor
                        ↪ atorBase<ulong>)links.Unsync).Links;
175                     using (var transaction = transactionsLayer.BeginTransaction())
176                     {
177                         var l1 = links.CreateAndUpdate(itself, itself);
178                         var l2 = links.CreateAndUpdate(itself, itself);
179
180                         l2 = links.Update(l2, l2, l1, l2);
181
182                         links.CreateAndUpdate(l2, itself);
183                         links.CreateAndUpdate(l2, itself);
184
185                         //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi
                            ↪ tion>(scope.TempTransactionLogFilename);
186
187                         l2 = links.Update(l2, l1);
188
189                         links.Delete(l2);
190
```

```
                    ExceptionThrower();

                    transaction.Commit();
                }

                Global.Trash = links.Count();
            }
        }
        catch
        {
            Assert.False(lastScope == null);

            var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(lastScope.TempTransactionLogFilename);

            Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() && transitions[0].After.IsNull());

            lastScope.DeleteFiles();
        }
    }

    [Fact]
    public static void TransactionUserCodeErrorSomeDataSavedTest()
    {
        // User Code Error (Autoreverted), some data saved
        var itself = _constants.Itself;

        TempLinksTestScope lastScope = null;
        try
        {
            ulong l1;
            ulong l2;

            using (var scope = new TempLinksTestScope(useLog: true))
            {
                var links = scope.Links;
                l1 = links.CreateAndUpdate(itself, itself);
                l2 = links.CreateAndUpdate(itself, itself);

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

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

                links.Unsync.DisposeIfPossible();

                Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope.TempTransactionLogFilename);
            }

            using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false, useLog: true))
            {
                var links = scope.Links;
                var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
                using (var transaction = transactionsLayer.BeginTransaction())
                {
                    l2 = links.Update(l2, l1);

                    links.Delete(l2);

                    ExceptionThrower();

                    transaction.Commit();
                }

                Global.Trash = links.Count();
            }
        }
        catch
        {
            Assert.False(lastScope == null);

            Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(lastScope.TempTransactionLogFilename);

            lastScope.DeleteFiles();
        }
    }
```

```
266
267         [Fact]
268         public static void TransactionCommit()
269         {
270             var itself = _constants.Itself;
271
272             var tempDatabaseFilename = Path.GetTempFileName();
273             var tempTransactionLogFilename = Path.GetTempFileName();
274
275             // Commit
276             using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                ↪  UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                ↪  tempTransactionLogFilename))
277             using (var links = new UInt64Links(memoryAdapter))
278             {
279                 using (var transaction = memoryAdapter.BeginTransaction())
280                 {
281                     var l1 = links.CreateAndUpdate(itself, itself);
282                     var l2 = links.CreateAndUpdate(itself, itself);
283
284                     Global.Trash = links.Update(l2, l2, l1, l2);
285
286                     links.Delete(l1);
287
288                     transaction.Commit();
289                 }
290
291                 Global.Trash = links.Count();
292             }
293
294             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                ↪  sactionLogFilename);
295         }
296
297         [Fact]
298         public static void TransactionDamage()
299         {
300             var itself = _constants.Itself;
301
302             var tempDatabaseFilename = Path.GetTempFileName();
303             var tempTransactionLogFilename = Path.GetTempFileName();
304
305             // Commit
306             using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                ↪  UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                ↪  tempTransactionLogFilename))
307             using (var links = new UInt64Links(memoryAdapter))
308             {
309                 using (var transaction = memoryAdapter.BeginTransaction())
310                 {
311                     var l1 = links.CreateAndUpdate(itself, itself);
312                     var l2 = links.CreateAndUpdate(itself, itself);
313
314                     Global.Trash = links.Update(l2, l2, l1, l2);
315
316                     links.Delete(l1);
317
318                     transaction.Commit();
319                 }
320
321                 Global.Trash = links.Count();
322             }
323
324             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                ↪  sactionLogFilename);
325
326             // Damage database
327
328             FileHelpers.WriteFirst(tempTransactionLogFilename, new
                ↪  UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
329
330             // Try load damaged database
331             try
332             {
333                 // TODO: Fix
334                 using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                    ↪  UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                    ↪  tempTransactionLogFilename))
335                 using (var links = new UInt64Links(memoryAdapter))
```

```
336                 {
337                     Global.Trash = links.Count();
338                 }
339             }
340             catch (NotSupportedException ex)
341             {
342                 Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
                    ↪ yet.");
343             }
344 
345             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                ↪ sactionLogFilename);
346 
347             File.Delete(tempDatabaseFilename);
348             File.Delete(tempTransactionLogFilename);
349         }
350 
351         [Fact]
352         public static void Bug1Test()
353         {
354             var tempDatabaseFilename = Path.GetTempFileName();
355             var tempTransactionLogFilename = Path.GetTempFileName();
356 
357             var itself = _constants.Itself;
358 
359             // User Code Error (Autoreverted), some data saved
360             try
361             {
362                 ulong l1;
363                 ulong l2;
364 
365                 using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                    ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                    ↪ tempTransactionLogFilename))
366                 using (var links = new UInt64Links(memoryAdapter))
367                 {
368                     l1 = links.CreateAndUpdate(itself, itself);
369                     l2 = links.CreateAndUpdate(itself, itself);
370 
371                     l2 = links.Update(l2, l2, l1, l2);
372 
373                     links.CreateAndUpdate(l2, itself);
374                     links.CreateAndUpdate(l2, itself);
375                 }
376 
377                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
                    ↪ TransactionLogFilename);
378 
379                 using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                    ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                    ↪ tempTransactionLogFilename))
380                 using (var links = new UInt64Links(memoryAdapter))
381                 {
382                     using (var transaction = memoryAdapter.BeginTransaction())
383                     {
384                         l2 = links.Update(l2, l1);
385 
386                         links.Delete(l2);
387 
388                         ExceptionThrower();
389 
390                         transaction.Commit();
391                     }
392 
393                     Global.Trash = links.Count();
394                 }
395             }
396             catch
397             {
398                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
                    ↪ TransactionLogFilename);
399             }
400 
401             File.Delete(tempDatabaseFilename);
402             File.Delete(tempTransactionLogFilename);
403         }
404 
405         private static void ExceptionThrower()
406         {
```

```
407             throw new Exception();
408         }
409
410         [Fact]
411         public static void PathsTest()
412         {
413             var source = _constants.SourcePart;
414             var target = _constants.TargetPart;
415
416             using (var scope = new TempLinksTestScope())
417             {
418                 var links = scope.Links;
419                 var l1 = links.CreatePoint();
420                 var l2 = links.CreatePoint();
421
422                 var r1 = links.GetByKeys(l1, source, target, source);
423                 var r2 = links.CheckPathExistance(l2, l2, l2, l2);
424             }
425         }
426
427         [Fact]
428         public static void RecursiveStringFormattingTest()
429         {
430             using (var scope = new TempLinksTestScope(useSequences: true))
431             {
432                 var links = scope.Links;
433                 var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.
434
435                 var a = links.CreatePoint();
436                 var b = links.CreatePoint();
437                 var c = links.CreatePoint();
438
439                 var ab = links.CreateAndUpdate(a, b);
440                 var cb = links.CreateAndUpdate(c, b);
441                 var ac = links.CreateAndUpdate(a, c);
442
443                 a = links.Update(a, c, b);
444                 b = links.Update(b, a, c);
445                 c = links.Update(c, a, b);
446
447                 Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
448                 Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
449                 Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
450
451                 Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
                    ↪  "(5:(4:5 (6:5 4)) 6)");
452                 Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
                    ↪  "(6:(5:(4:5 6) 6) 4)");
453                 Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
                    ↪  "(4:(5:4 (6:5 4)) 6)");
454
455                 // TODO: Think how to build balanced syntax tree while formatting structure (eg.
                    ↪  "(4:(5:4 6) (6:5 4)") instead of "(4:(5:4 (6:5 4)) 6)"
456
457                 Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
                    ↪  "{{5}{5}{4}{6}}");
458                 Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
                    ↪  "{{5}{6}{6}{4}}");
459                 Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
                    ↪  "{{4}{5}{4}{6}}");
460             }
461         }
462
463         private static void DefaultFormatter(StringBuilder sb, ulong link)
464         {
465             sb.Append(link.ToString());
466         }
467
468         #endregion
469
470         #region Performance
471
472         /*
473         public static void RunAllPerformanceTests()
474         {
475             try
476             {
477                 links.TestLinksInSteps();
478             }
```

```
            catch (Exception ex)
            {
                ex.WriteToConsole();
            }

            return;

            try
            {
                //ThreadPool.SetMaxThreads(2, 2);

                // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
↪  результат
                // Также это дополнительно помогает в отладке
                // Увеличивает вероятность попадания информации в кэши
                for (var i = 0; i < 10; i++)
                {
                    //0 - 10 ГБ
                    //Каждые 100 МБ срез цифр

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

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

                links.TestDeletionOfAllLinks();

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

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

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

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

            GetBaseRandomLoopOverhead(linksStep);
            GetBaseRandomLoopOverhead(linksStep);

            var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);

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

            var loops = totalLinksToCreate / linksStep;

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

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

            ConsoleHelpers.Debug();

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

```

```
556                 Console.Write("\rD {0}/{1}", i + 1, loops);
557             }
558 
559             ConsoleHelpers.Debug();
560 
561             ConsoleHelpers.Debug("C S D");
562 
563             for (int i = 0; i < loops; i++)
564             {
565                 ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
    ↪  searchMeasuremets[i], deletionMeasurements[i]);
566             }
567 
568             ConsoleHelpers.Debug("C S D (no overhead)");
569 
570             for (int i = 0; i < loops; i++)
571             {
572                 ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] - stepLoopOverhead,
    ↪  searchMeasuremets[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
573             }
574 
575             ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
    ↪  links.Total);
576         }
577 
578         private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
    ↪  amountToCreate)
579         {
580             for (long i = 0; i < amountToCreate; i++)
581                 links.Create(0, 0);
582         }
583 
584          private static TimeSpan GetBaseRandomLoopOverhead(long loops)
585          {
586              return Measure(() =>
587              {
588                  ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
589                  ulong result = 0;
590                  for (long i = 0; i < loops; i++)
591                  {
592                      var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
593                      var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
594 
595                      result += maxValue + source + target;
596                  }
597                  Global.Trash = result;
598              });
599          }
600           */
601 
602          [Fact(Skip = "performance test")]
603          public static void GetSourceTest()
604          {
605              using (var scope = new TempLinksTestScope())
606              {
607                  var links = scope.Links;
608                  ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
                     ↪  Iterations);
609 
610                  ulong counter = 0;
611 
612                  //var firstLink = links.First();
613                  // Создаём одну связь, из которой будет производить считывание
614                  var firstLink = links.Create();
615 
616                  var sw = Stopwatch.StartNew();
617 
618                  // Тестируем саму функцию
619                  for (ulong i = 0; i < Iterations; i++)
620                  {
621                      counter += links.GetSource(firstLink);
622                  }
623 
624                  var elapsedTime = sw.Elapsed;
625 
626                  var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
627 
628                  // Удаляем связь, из которой производилось считывание
629                  links.Delete(firstLink);
```

```
630
631                 ConsoleHelpers.Debug(
632                     "{0} Iterations of GetSource function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
633                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
634             }
635         }
636
637         [Fact(Skip = "performance test")]
638         public static void GetSourceInParallel()
639         {
640             using (var scope = new TempLinksTestScope())
641             {
642                 var links = scope.Links;
643                 ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
                    ↪  parallel.", Iterations);
644
645                 long counter = 0;
646
647                 //var firstLink = links.First();
648                 var firstLink = links.Create();
649
650                 var sw = Stopwatch.StartNew();
651
652                 // Тестируем саму функцию
653                 Parallel.For(0, Iterations, x =>
654                 {
655                     Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
656                     //Interlocked.Increment(ref counter);
657                 });
658
659                 var elapsedTime = sw.Elapsed;
660
661                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
662
663                 links.Delete(firstLink);
664
665                 ConsoleHelpers.Debug(
666                     "{0} Iterations of GetSource function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
667                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
668             }
669         }
670
671         [Fact(Skip = "performance test")]
672         public static void TestGetTarget()
673         {
674             using (var scope = new TempLinksTestScope())
675             {
676                 var links = scope.Links;
677                 ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
                    ↪  Iterations);
678
679                 ulong counter = 0;
680
681                 //var firstLink = links.First();
682                 var firstLink = links.Create();
683
684                 var sw = Stopwatch.StartNew();
685
686                 for (ulong i = 0; i < Iterations; i++)
687                 {
688                     counter += links.GetTarget(firstLink);
689                 }
690
691                 var elapsedTime = sw.Elapsed;
692
693                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
694
695                 links.Delete(firstLink);
696
697                 ConsoleHelpers.Debug(
698                     "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
699                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
700             }
701         }
702
703         [Fact(Skip = "performance test")]
```

```
704         public static void TestGetTargetInParallel()
705         {
706             using (var scope = new TempLinksTestScope())
707             {
708                 var links = scope.Links;
709                 ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
                    ↪  parallel.", Iterations);
710 
711                 long counter = 0;
712 
713                 //var firstLink = links.First();
714                 var firstLink = links.Create();
715 
716                 var sw = Stopwatch.StartNew();
717 
718                 Parallel.For(0, Iterations, x =>
719                 {
720                     Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
721                     //Interlocked.Increment(ref counter);
722                 });
723 
724                 var elapsedTime = sw.Elapsed;
725 
726                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
727 
728                 links.Delete(firstLink);
729 
730                 ConsoleHelpers.Debug(
731                     "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
732                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
733             }
734         }
735 
736         // TODO: Заполнить базу данных перед тестом
737         /*
738         [Fact]
739         public void TestRandomSearchFixed()
740         {
741             var tempFilename = Path.GetTempFileName();
742 
743             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪  DefaultLinksSizeStep))
744             {
745                 long iterations = 64 * 1024 * 1024 /
    ↪  Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
746 
747                 ulong counter = 0;
748                 var maxLink = links.Total;
749 
750                 ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);
751 
752                 var sw = Stopwatch.StartNew();
753 
754                 for (var i = iterations; i > 0; i--)
755                 {
756                     var source =
    ↪  RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
757                     var target =
    ↪  RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
758 
759                     counter += links.Search(source, target);
760                 }
761 
762                 var elapsedTime = sw.Elapsed;
763 
764                 var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
765 
766                 ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
    ↪  Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
    ↪  counter);
767             }
768 
769             File.Delete(tempFilename);
770         }*/
771 
772         [Fact(Skip = "useless: O(0), was dependent on creation tests")]
773         public static void TestRandomSearchAll()
774         {
```

```
775             using (var scope = new TempLinksTestScope())
776             {
777                 var links = scope.Links;
778                 ulong counter = 0;
779 
780                 var maxLink = links.Count();
781 
782                 var iterations = links.Count();
783 
784                 ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
                    ↪  links.Count());
785 
786                 var sw = Stopwatch.StartNew();
787 
788                 for (var i = iterations; i > 0; i--)
789                 {
790                     var linksAddressRange = new Range<ulong>(_constants.MinPossibleIndex,
                        ↪  maxLink);
791 
792                     var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
793                     var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
794 
795                     counter += links.SearchOrDefault(source, target);
796                 }
797 
798                 var elapsedTime = sw.Elapsed;
799 
800                 var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
801 
802                 ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
                    ↪  Iterations per second), c: {3}",
803                      iterations, elapsedTime, (long)iterationsPerSecond, counter);
804             }
805         }
806 
807         [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
808         public static void TestEach()
809         {
810             using (var scope = new TempLinksTestScope())
811             {
812                 var links = scope.Links;
813 
814                 var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
815 
816                 ConsoleHelpers.Debug("Testing Each function.");
817 
818                 var sw = Stopwatch.StartNew();
819 
820                 links.Each(counter.IncrementAndReturnTrue);
821 
822                 var elapsedTime = sw.Elapsed;
823 
824                 var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
825 
826                 ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
                    ↪  links per second)",
827                     counter, elapsedTime, (long)linksPerSecond);
828             }
829         }
830 
831         /*
832         [Fact]
833         public static void TestForeach()
834         {
835             var tempFilename = Path.GetTempFileName();
836 
837             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪  DefaultLinksSizeStep))
838             {
839                 ulong counter = 0;
840 
841                 ConsoleHelpers.Debug("Testing foreach through links.");
842 
843                 var sw = Stopwatch.StartNew();
844 
845                 //foreach (var link in links)
846                 //{
847                 //    counter++;
848                 //}
849 
```

```
850                 var elapsedTime = sw.Elapsed;
851
852                 var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
853
854                 ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
    ↪  links per second)", counter, elapsedTime, (long)linksPerSecond);
855             }
856
857             File.Delete(tempFilename);
858         }
859         */
860
861         /*
862         [Fact]
863         public static void TestParallelForeach()
864         {
865             var tempFilename = Path.GetTempFileName();
866
867             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪  DefaultLinksSizeStep))
868             {
869
870                 long counter = 0;
871
872                 ConsoleHelpers.Debug("Testing parallel foreach through links.");
873
874                 var sw = Stopwatch.StartNew();
875
876                 //Parallel.ForEach((IEnumerable<ulong>)links, x =>
877                 //{
878                 //    Interlocked.Increment(ref counter);
879                 //});
880
881                 var elapsedTime = sw.Elapsed;
882
883                 var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
884
885                 ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
    ↪  {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
886             }
887
888             File.Delete(tempFilename);
889         }
890         */
891
892         [Fact(Skip = "performance test")]
893         public static void Create64BillionLinks()
894         {
895             using (var scope = new TempLinksTestScope())
896             {
897                 var links = scope.Links;
898                 var linksBeforeTest = links.Count();
899
900                 long linksToCreate = 64 * 1024 * 1024 /
                    ↪  UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
901
902                 ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
903
904                 var elapsedTime = Performance.Measure(() =>
905                 {
906                     for (long i = 0; i < linksToCreate; i++)
907                     {
908                         links.Create();
909                     }
910                 });
911
912                 var linksCreated = links.Count() - linksBeforeTest;
913                 var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
914
915                 ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
916
917                 ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
                    ↪  linksCreated, elapsedTime,
918                     (long)linksPerSecond);
919             }
920         }
921
922         [Fact(Skip = "performance test")]
923         public static void Create64BillionLinksInParallel()
```

```
924         {
925             using (var scope = new TempLinksTestScope())
926             {
927                 var links = scope.Links;
928                 var linksBeforeTest = links.Count();
929 
930                 var sw = Stopwatch.StartNew();
931 
932                 long linksToCreate = 64 * 1024 * 1024 /
                    ↪ UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
933 
934                 ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);
935 
936                 Parallel.For(0, linksToCreate, x => links.Create());
937 
938                 var elapsedTime = sw.Elapsed;
939 
940                 var linksCreated = links.Count() - linksBeforeTest;
941                 var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
942 
943                 ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
                    ↪ linksCreated, elapsedTime,
944                     (long)linksPerSecond);
945             }
946         }
947 
948         [Fact(Skip = "useless: O(0), was dependent on creation tests")]
949         public static void TestDeletionOfAllLinks()
950         {
951             using (var scope = new TempLinksTestScope())
952             {
953                 var links = scope.Links;
954                 var linksBeforeTest = links.Count();
955 
956                 ConsoleHelpers.Debug("Deleting all links");
957 
958                 var elapsedTime = Performance.Measure(links.DeleteAll);
959 
960                 var linksDeleted = linksBeforeTest - links.Count();
961                 var linksPerSecond = linksDeleted / elapsedTime.TotalSeconds;
962 
963                 ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
                    ↪ linksDeleted, elapsedTime,
964                     (long)linksPerSecond);
965             }
966         }
967 
968         #endregion
969     }
970 }
```

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

```
1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4  using Xunit;
5  using Platform.Interfaces;
6  using Platform.Data.Doublets.Sequences;
7  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
8  using Platform.Data.Doublets.Sequences.Frequencies.Counters;
9  using Platform.Data.Doublets.Sequences.Converters;
10 using Platform.Data.Doublets.PropertyOperators;
11 using Platform.Data.Doublets.Incrementers;
12 using Platform.Data.Doublets.Converters;
13 
14 namespace Platform.Data.Doublets.Tests
15 {
16     public static class OptimalVariantSequenceTests
17     {
18         private const string SequenceExample = "зеленела зелёная зелень";
19 
20         [Fact]
21         public static void LinksBasedFrequencyStoredOptimalVariantSequenceTest()
22         {
23             using (var scope = new TempLinksTestScope(useSequences: true))
24             {
25                 var links = scope.Links;
26                 var sequences = scope.Sequences;
27                 var constants = links.Constants;
28 
29                 links.UseUnicode();
```

```
30
31                  var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);
32
33                  var meaningRoot = links.CreatePoint();
34                  var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
35                  var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
36                  var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
                    ↪  constants.Itself);
37
38                  var unaryNumberToAddressConveter = new
                    ↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
39                  var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
40                  var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                    ↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
41                  var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
                    ↪  frequencyPropertyMarker, frequencyMarker);
42                  var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
                    ↪  frequencyPropertyOperator, frequencyIncrementer);
43                  var linkToItsFrequencyNumberConverter = new
                    ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                    ↪  unaryNumberToAddressConveter);
44                  var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                    ↪  linkToItsFrequencyNumberConverter);
45                  var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                    ↪  sequenceToItsLocalElementLevelsConverter);
46
47                  ExecuteTest(links, sequences, sequence,
                    ↪  sequenceToItsLocalElementLevelsConverter, linkFrequencyIncrementer,
                    ↪  optimalVariantConverter);
48              }
49          }
50
51          [Fact]
52          public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
53          {
54              using (var scope = new TempLinksTestScope(useSequences: true))
55              {
56                  var links = scope.Links;
57                  var sequences = scope.Sequences;
58
59                  links.UseUnicode();
60
61                  var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);
62
63                  var linksToFrequencies = new Dictionary<ulong, ulong>();
64
65                  var totalSequenceSymbolFrequencyCounter = new
                    ↪  TotalSequenceSymbolFrequencyCounter<ulong>(links);
66
67                  var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
                    ↪  totalSequenceSymbolFrequencyCounter);
68
69                  var linkFrequencyIncrementer = new
                    ↪  FrequenciesCacheBasedLinkFrequencyIncrementer<ulong>(linkFrequenciesCache);
70                  var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
                    ↪  ncyNumberConverter<ulong>(linkFrequenciesCache);
71
72                  var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                    ↪  linkToItsFrequencyNumberConverter);
73                  var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                    ↪  sequenceToItsLocalElementLevelsConverter);
74
75                  ExecuteTest(links, sequences, sequence,
                    ↪  sequenceToItsLocalElementLevelsConverter, linkFrequencyIncrementer,
                    ↪  optimalVariantConverter);
76              }
77          }
78
79          private static void ExecuteTest(SynchronizedLinks<ulong> links, Sequences.Sequences
            ↪  sequences, ulong[] sequence, SequenceToItsLocalElementLevelsConverter<ulong>
            ↪  sequenceToItsLocalElementLevelsConverter, IIncrementer<IList<ulong>>
            ↪  linkFrequencyIncrementer, OptimalVariantConverter<ulong> optimalVariantConverter)
80          {
81              linkFrequencyIncrementer.Increment(sequence);
82
83              var levels = sequenceToItsLocalElementLevelsConverter.Convert(sequence);
```

```

            var optimalVariant = optimalVariantConverter.Convert(sequence);

            var readSequence1 = sequences.ReadSequenceCore(optimalVariant, links.IsPartialPoint);

            Assert.True(sequence.SequenceEqual(readSequence1));
        }
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using Xunit;
using Platform.Data.Sequences;
using Platform.Data.Doublets.Sequences.Converters;

namespace Platform.Data.Doublets.Tests
{
    public static class ReadSequenceTests
    {
        [Fact]
        public static void ReadSequenceTest()
        {
            const long sequenceLength = 2000;

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

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

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

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

                var sw2 = Stopwatch.StartNew();
                var readSequence1 = sequences.ReadSequenceCore(balancedVariant,
                ↪  links.IsPartialPoint); sw2.Stop();

                var sw3 = Stopwatch.StartNew();
                var readSequence2 = new List<ulong>();
                SequenceWalker.WalkRight(balancedVariant,
                                         links.GetSource,
                                         links.GetTarget,
                                         links.IsPartialPoint,
                                         readSequence2.Add);
                sw3.Stop();

                Assert.True(sequence.SequenceEqual(readSequence1));

                Assert.True(sequence.SequenceEqual(readSequence2));

                // Assert.True(sw2.Elapsed < sw3.Elapsed);

                Console.WriteLine($"Stack-based walker: {sw3.Elapsed}, Level-based reader:
                ↪  {sw2.Elapsed}");

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

./Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs

```
using System.IO;
using Xunit;
using Platform.Singletons;
using Platform.Memory;
```

```
using Platform.Data.Constants;
using Platform.Data.Doublets.ResizableDirectMemory;

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

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

            using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(tempFilename))
            {
                memoryAdapter.TestBasicMemoryOperations();
            }

            File.Delete(tempFilename);
        }

        [Fact]
        public static void BasicHeapMemoryTest()
        {
            using (var memory = new
            ↪  HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
            using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
            ↪  UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
            {
                memoryAdapter.TestBasicMemoryOperations();
            }
        }

        private static void TestBasicMemoryOperations(this ILinks<ulong> memoryAdapter)
        {
            var link = memoryAdapter.Create();
            memoryAdapter.Delete(link);
        }

        [Fact]
        public static void NonexistentReferencesHeapMemoryTest()
        {
            using (var memory = new
            ↪  HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
            using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
            ↪  UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
            {
                memoryAdapter.TestNonexistentReferences();
            }
        }

        private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
        {
            var link = memoryAdapter.Create();

            memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);

            var resultLink = _constants.Null;

            memoryAdapter.Each(foundLink =>
            {
                resultLink = foundLink[_constants.IndexPart];
                return _constants.Break;
            }, _constants.Any, ulong.MaxValue, ulong.MaxValue);

            Assert.True(resultLink == link);

            Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);

            memoryAdapter.Delete(link);
        }
    }
}
```

./Platform.Data.Doublets.Tests/ScopeTests.cs

```
using Xunit;
using Platform.Scopes;
```

```
using Platform.Memory;
using Platform.Data.Doublets.ResizableDirectMemory;
using Platform.Data.Doublets.Decorators;

namespace Platform.Data.Doublets.Tests
{
    public static class ScopeTests
    {
        [Fact]
        public static void SingleDependencyTest()
        {
            using (var scope = new Scope())
            {
                scope.IncludeAssemblyOf<IMemory>();
                var instance = scope.Use<IDirectMemory>();
                Assert.IsType<HeapResizableDirectMemory>(instance);
            }
        }

        [Fact]
        public static void CascadeDependencyTest()
        {
            using (var scope = new Scope())
            {
                scope.Include<TemporaryFileMappedResizableDirectMemory>();
                scope.Include<UInt64ResizableDirectMemoryLinks>();
                var instance = scope.Use<ILinks<ulong>>();
                Assert.IsType<UInt64ResizableDirectMemoryLinks>(instance);
            }
        }

        [Fact]
        public static void FullAutoResolutionTest()
        {
            using (var scope = new Scope(autoInclude: true, autoExplore: true))
            {
                var instance = scope.Use<UInt64Links>();
                Assert.IsType<UInt64Links>(instance);
            }
        }
    }
}
```

./Platform.Data.Doublets.Tests/SequencesTests.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using Xunit;
using Platform.Collections;
using Platform.Random;
using Platform.IO;
using Platform.Singletons;
using Platform.Data.Constants;
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Converters;

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

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

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

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

```
            var sequences = scope.Sequences;

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

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

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

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

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

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

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

    //    const long sequenceLength = 8;

    //    const ulong itself = LinksConstants.Itself;

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


    //        SequencesOptions o = new SequencesOptions();

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


    //        var sequences = new Sequences(links);

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

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

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

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

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

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

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

            var sequence = new ulong[sequenceLength];
            for (var i = 0; i < sequenceLength; i++)
```

```
115                 {
116                     sequence[i] = links.Create();
117                 }
118 
119                 var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();
120 
121                 //for (int i = 0; i < createResults.Length; i++)
122                 //    sequences.Create(createResults[i]);
123 
124                 var sw0 = Stopwatch.StartNew();
125                 var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
126 
127                 var sw1 = Stopwatch.StartNew();
128                 var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
129 
130                 var sw2 = Stopwatch.StartNew();
131                 var searchResults2 = sequences.Each1(sequence); sw2.Stop();
132 
133                 var sw3 = Stopwatch.StartNew();
134                 var searchResults3 = sequences.Each(sequence); sw3.Stop();
135 
136                 var intersection0 = createResults.Intersect(searchResults0).ToList();
137                 Assert.True(intersection0.Count == searchResults0.Count);
138                 Assert.True(intersection0.Count == createResults.Length);
139 
140                 var intersection1 = createResults.Intersect(searchResults1).ToList();
141                 Assert.True(intersection1.Count == searchResults1.Count);
142                 Assert.True(intersection1.Count == createResults.Length);
143 
144                 var intersection2 = createResults.Intersect(searchResults2).ToList();
145                 Assert.True(intersection2.Count == searchResults2.Count);
146                 Assert.True(intersection2.Count == createResults.Length);
147 
148                 var intersection3 = createResults.Intersect(searchResults3).ToList();
149                 Assert.True(intersection3.Count == searchResults3.Count);
150                 Assert.True(intersection3.Count == createResults.Length);
151 
152                 for (var i = 0; i < sequenceLength; i++)
153                 {
154                     links.Delete(sequence[i]);
155                 }
156             }
157         }
158 
159         [Fact]
160         public static void BalancedVariantSearchTest()
161         {
162             const long sequenceLength = 200;
163 
164             using (var scope = new TempLinksTestScope(useSequences: true))
165             {
166                 var links = scope.Links;
167                 var sequences = scope.Sequences;
168 
169                 var sequence = new ulong[sequenceLength];
170                 for (var i = 0; i < sequenceLength; i++)
171                 {
172                     sequence[i] = links.Create();
173                 }
174 
175                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
176 
177                 var sw1 = Stopwatch.StartNew();
178                 var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
179 
180                 var sw2 = Stopwatch.StartNew();
181                 var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();
182 
183                 var sw3 = Stopwatch.StartNew();
184                 var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();
185 
186                 // На количестве в 200 элементов это будет занимать вечность
187                 //var sw4 = Stopwatch.StartNew();
188                 //var searchResults4 = sequences.Each(sequence); sw4.Stop();
189 
190                 Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);
191 
192                 Assert.True(searchResults3.Count == 1 && balancedVariant ==
                    ↪ searchResults3.First());
193 
```

```
194                 //Assert.True(sw1.Elapsed < sw2.Elapsed);
195
196                 for (var i = 0; i < sequenceLength; i++)
197                 {
198                     links.Delete(sequence[i]);
199                 }
200             }
201         }
202
203         [Fact]
204         public static void AllPartialVariantsSearchTest()
205         {
206             const long sequenceLength = 8;
207
208             using (var scope = new TempLinksTestScope(useSequences: true))
209             {
210                 var links = scope.Links;
211                 var sequences = scope.Sequences;
212
213                 var sequence = new ulong[sequenceLength];
214                 for (var i = 0; i < sequenceLength; i++)
215                 {
216                     sequence[i] = links.Create();
217                 }
218
219                 var createResults = sequences.CreateAllVariants2(sequence);
220
221                 //var createResultsStrings = createResults.Select(x => x + ": " +
                    ↪ sequences.FormatSequence(x)).ToList();
222                 //Global.Trash = createResultsStrings;
223
224                 var partialSequence = new ulong[sequenceLength - 2];
225
226                 Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
227
228                 var sw1 = Stopwatch.StartNew();
229                 var searchResults1 =
                    ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
230
231                 var sw2 = Stopwatch.StartNew();
232                 var searchResults2 =
                    ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
233
234                 //var sw3 = Stopwatch.StartNew();
235                 //var searchResults3 =
                    ↪ sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
236
237                 var sw4 = Stopwatch.StartNew();
238                 var searchResults4 =
                    ↪ sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();
239
240                 //Global.Trash = searchResults3;
241
242                 //var searchResults1Strings = searchResults1.Select(x => x + ": " +
                    ↪ sequences.FormatSequence(x)).ToList();
243                 //Global.Trash = searchResults1Strings;
244
245                 var intersection1 = createResults.Intersect(searchResults1).ToList();
246                 Assert.True(intersection1.Count == createResults.Length);
247
248                 var intersection2 = createResults.Intersect(searchResults2).ToList();
249                 Assert.True(intersection2.Count == createResults.Length);
250
251                 var intersection4 = createResults.Intersect(searchResults4).ToList();
252                 Assert.True(intersection4.Count == createResults.Length);
253
254                 for (var i = 0; i < sequenceLength; i++)
255                 {
256                     links.Delete(sequence[i]);
257                 }
258             }
259         }
260
261         [Fact]
262         public static void BalancedPartialVariantsSearchTest()
263         {
264             const long sequenceLength = 200;
265
266             using (var scope = new TempLinksTestScope(useSequences: true))
```

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

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

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

            var balancedVariant = balancedVariantConverter.Convert(sequence);

            var partialSequence = new ulong[sequenceLength - 2];

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

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

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

            Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);

            Assert.True(searchResults2.Count == 1 && balancedVariant ==
            ↪  searchResults2.First());

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

    [Fact(Skip = "Correct implementation is pending")]
    public static void PatternMatchTest()
    {
        var zeroOrMany = Sequences.Sequences.ZeroOrMany;

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

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

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

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

            var balancedVariant = balancedVariantConverter.Convert(sequence);

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

            var doublet = links.GetSource(balancedVariant);

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

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

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

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

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

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

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

```
344
345                 Assert.Contains(doublet, matchedSequences4);
346                 Assert.Contains(balancedVariant, matchedSequences4);
347
348                 for (var i = 0; i < sequence.Length; i++)
349                 {
350                     links.Delete(sequence[i]);
351                 }
352             }
353         }
354
355         [Fact]
356         public static void IndexTest()
357         {
358             using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
                ↪ true }, useSequences: true))
359             {
360                 var links = scope.Links;
361                 var sequences = scope.Sequences;
362                 var indexer = sequences.Options.Indexer;
363
364                 var e1 = links.Create();
365                 var e2 = links.Create();
366
367                 var sequence = new[]
368                 {
369                     e1, e2, e1, e2 // mama / papa
370                 };
371
372                 Assert.False(indexer.Index(sequence));
373
374                 Assert.True(indexer.Index(sequence));
375             }
376         }
377
378         /// <summary>Imported from https://raw.githubusercontent.com/wiki/Konard/LinksPlatform/%
            ↪ D0%9E-%D1%82%D0%BE%D0%BC%2C-%D0%BA%D0%B0%D0%BA-%D0%B2%D1%81%D1%91-%D0%BD%D0%B0%D1%87
            ↪ %D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>
379         private static readonly string _exampleText =
380             @"([english
                ↪ version](https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))
381
382 Обозначение пустоты, какое оно? Темнота ли это? Там где отсутствие света, отсутствие фотонов
    ↪ (носителей света)? Или это то, что полностью отражает свет? Пустой белый лист бумаги? Там
    ↪ где есть место для нового начала? Разве пустота это не характеристика пространства?
    ↪ Пространство это то, что можно чем-то наполнить?
383
384 [![чёрное пространство, белое
    ↪ пространство](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png
    ↪ ""чёрное пространство, белое пространство"")](https://raw.githubusercontent.com/Konard/Links
    ↪ Platform/master/doc/Intro/1.png)
385
386 Что может быть минимальным рисунком, образом, графикой? Может быть это точка? Это ли простейшая
    ↪ форма? Но есть ли у точки размер? Цвет? Масса? Координаты? Время существования?
387
388 [![чёрное пространство, чёрная
    ↪ точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png
    ↪ ""чёрное пространство, чёрная
    ↪ точка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png)
389
390 А что если повторить? Сделать копию? Создать дубликат? Из одного сделать два? Может это быть
    ↪ так? Инверсия? Отражение? Сумма?
391
392 [![белая точка, чёрная
    ↪ точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png ""белая
    ↪ точка, чёрная
    ↪ точка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png)
393
394 А что если мы вообразим движение? Нужно ли время? Каким самым коротким будет путь? Что будет
    ↪ если этот путь зафиксировать? Запомнить след? Как две точки становятся линией? Чертой?
    ↪ Гранью? Разделителем? Единицей?
395
396 [![две белые точки, чёрная вертикальная
    ↪ линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png ""две
    ↪ белые точки, чёрная вертикальная
    ↪ линия"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png)
397
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

...

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


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

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

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

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

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

                var compressedVariant = compressingConverter.Convert(sequence);

                // 1: [1]       (1->1) point
                // 2: [2]       (2->2) point
                // 3: [1,2]     (1->2) doublet
                // 4: [1,2,1,2] (3->3) doublet

                Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
                Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
                Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
                Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);

                var source = _constants.SourcePart;
                var target = _constants.TargetPart;

                Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
                Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
                Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
                Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);

                // 4 - length of sequence
                Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0) == sequence[0]);
                Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1) == sequence[1]);
                Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2) == sequence[2]);
                Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3) == sequence[3]);
            }
        }

        [Fact]
        public static void CompressionEfficiencyTest()
```

```
490             {
491                 var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
                    ↪ StringSplitOptions.RemoveEmptyEntries);
492                 var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
493                 var totalCharacters = arrays.Select(x => x.Length).Sum();
494
495                 using (var scope1 = new TempLinksTestScope(useSequences: true))
496                 using (var scope2 = new TempLinksTestScope(useSequences: true))
497                 using (var scope3 = new TempLinksTestScope(useSequences: true))
498                 {
499                     scope1.Links.Unsync.UseUnicode();
500                     scope2.Links.Unsync.UseUnicode();
501                     scope3.Links.Unsync.UseUnicode();
502
503                     var balancedVariantConverter1 = new
                        ↪ BalancedVariantConverter<ulong>(scope1.Links.Unsync);
504                     var totalSequenceSymbolFrequencyCounter = new
                        ↪ TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
505                     var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
                        ↪ totalSequenceSymbolFrequencyCounter);
506                     var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
                        ↪ balancedVariantConverter1, linkFrequenciesCache1,
                        ↪ doInitialFrequenciesIncrement: false);
507
508                     var compressor2 = scope2.Sequences;
509                     var compressor3 = scope3.Sequences;
510
511                     var constants = Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
512
513                     var sequences = compressor3;
514                     //var meaningRoot = links.CreatePoint();
515                     //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
516                     //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
517                     //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
                        ↪ constants.Itself);
518
519                     //var unaryNumberToAddressConveter = new
                        ↪ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
520                     //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
                        ↪ unaryOne);
521                     //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                        ↪ frequencyMarker, unaryOne, unaryNumberIncrementer);
522                     //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
                        ↪ frequencyPropertyMarker, frequencyMarker);
523                     //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
                        ↪ frequencyPropertyOperator, frequencyIncrementer);
524                     //var linkToItsFrequencyNumberConverter = new
                        ↪ LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                        ↪ unaryNumberToAddressConveter);
525
526                     var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
                        ↪ totalSequenceSymbolFrequencyCounter);
527
528                     var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
                        ↪ ncyNumberConverter<ulong>(linkFrequenciesCache3);
529
530                     var sequenceToItsLocalElementLevelsConverter = new
                        ↪ SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
                        ↪ linkToItsFrequencyNumberConverter);
531                     var optimalVariantConverter = new
                        ↪ OptimalVariantConverter<ulong>(scope3.Links.Unsync,
                        ↪ sequenceToItsLocalElementLevelsConverter);
532
533                     var compressed1 = new ulong[arrays.Length];
534                     var compressed2 = new ulong[arrays.Length];
535                     var compressed3 = new ulong[arrays.Length];
536
537                     var START = 0;
538                     var END = arrays.Length;
539
540                     //for (int i = START; i < END; i++)
541                     //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
542
543                     var initialCount1 = scope2.Links.Unsync.Count();
544
545                     var sw1 = Stopwatch.StartNew();
546
547                     for (int i = START; i < END; i++)
```

```
548                 {
549                     linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
550                     compressed1[i] = compressor1.Convert(arrays[i]);
551                 }
552 
553                 var elapsed1 = sw1.Elapsed;
554 
555                 var balancedVariantConverter2 = new
                    ↪  BalancedVariantConverter<ulong>(scope2.Links.Unsync);
556 
557                 var initialCount2 = scope2.Links.Unsync.Count();
558 
559                 var sw2 = Stopwatch.StartNew();
560 
561                 for (int i = START; i < END; i++)
562                 {
563                     compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
564                 }
565 
566                 var elapsed2 = sw2.Elapsed;
567 
568                 for (int i = START; i < END; i++)
569                 {
570                     linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
571                 }
572 
573                 var initialCount3 = scope3.Links.Unsync.Count();
574 
575                 var sw3 = Stopwatch.StartNew();
576 
577                 for (int i = START; i < END; i++)
578                 {
579                     //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
580                     compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
581                 }
582 
583                 var elapsed3 = sw3.Elapsed;
584 
585                 Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
                    ↪  Optimal variant: {elapsed3}");
586 
587                 // Assert.True(elapsed1 > elapsed2);
588 
589                 // Checks
590                 for (int i = START; i < END; i++)
591                 {
592                     var sequence1 = compressed1[i];
593                     var sequence2 = compressed2[i];
594                     var sequence3 = compressed3[i];
595 
596                     var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                        ↪  scope1.Links.Unsync);
597 
598                     var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                        ↪  scope2.Links.Unsync);
599 
600                     var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
                        ↪  scope3.Links.Unsync);
601 
602                     var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
                        ↪  link.IsPartialPoint());
603                     var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
                        ↪  link.IsPartialPoint());
604                     var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
                        ↪  link.IsPartialPoint());
605 
606                     //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
                        ↪  arrays[i].Length > 3)
607                     //    Assert.False(structure1 == structure2);
608                     //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
                        ↪  arrays[i].Length > 3)
609                     //    Assert.False(structure3 == structure2);
610 
611                     Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
612                     Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
613                 }
614 
615                 Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
                    ↪  totalCharacters);
```

```
616                 Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
                    ↪ totalCharacters);
617                 Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
                    ↪ totalCharacters);
618
619                 Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
                    ↪ totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) /
                    ↪ totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) /
                    ↪ totalCharacters}");
620
621                 Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
                    ↪ scope2.Links.Unsync.Count() - initialCount2);
622                 Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
                    ↪ scope2.Links.Unsync.Count() - initialCount2);
623
624                 var duplicateProvider1 = new
                    ↪ DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
625                 var duplicateProvider2 = new
                    ↪ DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
626                 var duplicateProvider3 = new
                    ↪ DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
627
628                 var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
629                 var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
630                 var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
631
632                 var duplicates1 = duplicateCounter1.Count();
633
634                 ConsoleHelpers.Debug("------");
635
636                 var duplicates2 = duplicateCounter2.Count();
637
638                 ConsoleHelpers.Debug("------");
639
640                 var duplicates3 = duplicateCounter3.Count();
641
642                 Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
643
644                 linkFrequenciesCache1.ValidateFrequencies();
645                 linkFrequenciesCache3.ValidateFrequencies();
646             }
647         }
648
649         [Fact]
650         public static void CompressionStabilityTest()
651         {
652             // TODO: Fix bug (do a separate test)
653             //const ulong minNumbers = 0;
654             //const ulong maxNumbers = 1000;
655
656             const ulong minNumbers = 10000;
657             const ulong maxNumbers = 12500;
658
659             var strings = new List<string>();
660
661             for (ulong i = minNumbers; i < maxNumbers; i++)
662             {
663                 strings.Add(i.ToString());
664             }
665
666             var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
667             var totalCharacters = arrays.Select(x => x.Length).Sum();
668
669             using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
                    ↪ SequencesOptions<ulong> { UseCompression = true,
                    ↪ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
670             using (var scope2 = new TempLinksTestScope(useSequences: true))
671             {
672                 scope1.Links.UseUnicode();
673                 scope2.Links.UseUnicode();
674
675                 //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
676                 var compressor1 = scope1.Sequences;
677                 var compressor2 = scope2.Sequences;
678
679                 var compressed1 = new ulong[arrays.Length];
680                 var compressed2 = new ulong[arrays.Length];
681
682                 var sw1 = Stopwatch.StartNew();
```

```
683
684                 var START = 0;
685                 var END = arrays.Length;
686
687                 // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
688                 // Stability issue starts at 10001 or 11000
689                 //for (int i = START; i < END; i++)
690                 //{
691                 //    var first = compressor1.Compress(arrays[i]);
692                 //    var second = compressor1.Compress(arrays[i]);
693
694                 //    if (first == second)
695                 //        compressed1[i] = first;
696                 //    else
697                 //    {
698                 //        // TODO: Find a solution for this case
699                 //    }
700                 //}
701
702                 for (int i = START; i < END; i++)
703                 {
704                     var first = compressor1.Create(arrays[i]);
705                     var second = compressor1.Create(arrays[i]);
706
707                     if (first == second)
708                     {
709                         compressed1[i] = first;
710                     }
711                     else
712                     {
713                         // TODO: Find a solution for this case
714                     }
715                 }
716
717                 var elapsed1 = sw1.Elapsed;
718
719                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
720
721                 var sw2 = Stopwatch.StartNew();
722
723                 for (int i = START; i < END; i++)
724                 {
725                     var first = balancedVariantConverter.Convert(arrays[i]);
726                     var second = balancedVariantConverter.Convert(arrays[i]);
727
728                     if (first == second)
729                     {
730                         compressed2[i] = first;
731                     }
732                 }
733
734                 var elapsed2 = sw2.Elapsed;
735
736                 Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
                    {elapsed2}");
737
738                 Assert.True(elapsed1 > elapsed2);
739
740                 // Checks
741                 for (int i = START; i < END; i++)
742                 {
743                     var sequence1 = compressed1[i];
744                     var sequence2 = compressed2[i];
745
746                     if (sequence1 != _constants.Null && sequence2 != _constants.Null)
747                     {
748                         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                            scope1.Links);
749
750                         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                            scope2.Links);
751
752                         //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
                            link.IsPartialPoint());
753                         //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
                            link.IsPartialPoint());
754
755                         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
                            arrays[i].Length > 3)
```

```
                        //    Assert.False(structure1 == structure2);

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

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

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

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

                //compressor1.ValidateFrequencies();
            }
        }

        [Fact]
        public static void RundomNumbersCompressionQualityTest()
        {
            const ulong N = 500;

            //const ulong minNumbers = 10000;
            //const ulong maxNumbers = 20000;

            //var strings = new List<string>();

            //for (ulong i = 0; i < N; i++)
            //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
            ↪  maxNumbers).ToString());

            var strings = new List<string>();

            for (ulong i = 0; i < N; i++)
            {
                strings.Add(RandomHelpers.Default.NextUInt64().ToString());
            }

            strings = strings.Distinct().ToList();

            var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
            var totalCharacters = arrays.Select(x => x.Length).Sum();

            using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
            ↪  SequencesOptions<ulong> { UseCompression = true,
            ↪  EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
            using (var scope2 = new TempLinksTestScope(useSequences: true))
            {
                scope1.Links.UseUnicode();
                scope2.Links.UseUnicode();

                var compressor1 = scope1.Sequences;
                var compressor2 = scope2.Sequences;

                var compressed1 = new ulong[arrays.Length];
                var compressed2 = new ulong[arrays.Length];

                var sw1 = Stopwatch.StartNew();

                var START = 0;
                var END = arrays.Length;

                for (int i = START; i < END; i++)
                {
                    compressed1[i] = compressor1.Create(arrays[i]);
                }

                var elapsed1 = sw1.Elapsed;

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

                var sw2 = Stopwatch.StartNew();

                for (int i = START; i < END; i++)
                {
                    compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
                }

```

```
831                 var elapsed2 = sw2.Elapsed;
832
833                 Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
                    ↪ {elapsed2}");
834
835                 Assert.True(elapsed1 > elapsed2);
836
837                 // Checks
838                 for (int i = START; i < END; i++)
839                 {
840                     var sequence1 = compressed1[i];
841                     var sequence2 = compressed2[i];
842
843                     if (sequence1 != _constants.Null && sequence2 != _constants.Null)
844                     {
845                         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                            ↪ scope1.Links);
846
847                         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                            ↪ scope2.Links);
848
849                         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
850                     }
851                 }
852
853                 Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
854                 Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
855
856                 Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
                    ↪ totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
                    ↪ totalCharacters}");
857
858                 // Can be worse than balanced variant
859                 //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
860
861                 //compressor1.ValidateFrequencies();
862             }
863         }
864
865         [Fact]
866         public static void AllTreeBreakDownAtSequencesCreationBugTest()
867         {
868             // Made out of AllPossibleConnectionsTest test.
869
870             //const long sequenceLength = 5; //100% bug
871             const long sequenceLength = 4; //100% bug
872             //const long sequenceLength = 3; //100% _no_bug (ok)
873
874             using (var scope = new TempLinksTestScope(useSequences: true))
875             {
876                 var links = scope.Links;
877                 var sequences = scope.Sequences;
878
879                 var sequence = new ulong[sequenceLength];
880                 for (var i = 0; i < sequenceLength; i++)
881                 {
882                     sequence[i] = links.Create();
883                 }
884
885                 var createResults = sequences.CreateAllVariants2(sequence);
886
887                 Global.Trash = createResults;
888
889                 for (var i = 0; i < sequenceLength; i++)
890                 {
891                     links.Delete(sequence[i]);
892                 }
893             }
894         }
895
896         [Fact]
897         public static void AllPossibleConnectionsTest()
898         {
899             const long sequenceLength = 5;
900
901             using (var scope = new TempLinksTestScope(useSequences: true))
902             {
903                 var links = scope.Links;
904                 var sequences = scope.Sequences;
```

```
905
906                 var sequence = new ulong[sequenceLength];
907                 for (var i = 0; i < sequenceLength; i++)
908                 {
909                     sequence[i] = links.Create();
910                 }
911
912                 var createResults = sequences.CreateAllVariants2(sequence);
913                 var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
914
915                 for (var i = 0; i < 1; i++)
916                 {
917                     var sw1 = Stopwatch.StartNew();
918                     var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
919
920                     var sw2 = Stopwatch.StartNew();
921                     var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();
922
923                     var sw3 = Stopwatch.StartNew();
924                     var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();
925
926                     var sw4 = Stopwatch.StartNew();
927                     var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();
928
929                     Global.Trash = searchResults3;
930                     Global.Trash = searchResults4; //-V3008
931
932                     var intersection1 = createResults.Intersect(searchResults1).ToList();
933                     Assert.True(intersection1.Count == createResults.Length);
934
935                     var intersection2 = reverseResults.Intersect(searchResults1).ToList();
936                     Assert.True(intersection2.Count == reverseResults.Length);
937
938                     var intersection0 = searchResults1.Intersect(searchResults2).ToList();
939                     Assert.True(intersection0.Count == searchResults2.Count);
940
941                     var intersection3 = searchResults2.Intersect(searchResults3).ToList();
942                     Assert.True(intersection3.Count == searchResults3.Count);
943
944                     var intersection4 = searchResults3.Intersect(searchResults4).ToList();
945                     Assert.True(intersection4.Count == searchResults4.Count);
946                 }
947
948                 for (var i = 0; i < sequenceLength; i++)
949                 {
950                     links.Delete(sequence[i]);
951                 }
952             }
953         }
954
955         [Fact(Skip = "Correct implementation is pending")]
956         public static void CalculateAllUsagesTest()
957         {
958             const long sequenceLength = 3;
959
960             using (var scope = new TempLinksTestScope(useSequences: true))
961             {
962                 var links = scope.Links;
963                 var sequences = scope.Sequences;
964
965                 var sequence = new ulong[sequenceLength];
966                 for (var i = 0; i < sequenceLength; i++)
967                 {
968                     sequence[i] = links.Create();
969                 }
970
971                 var createResults = sequences.CreateAllVariants2(sequence);
972
973                 //var reverseResults =
                 ↪  sequences.CreateAllVariants2(sequence.Reverse().ToArray());
974
975                 for (var i = 0; i < 1; i++)
976                 {
977                     var linksTotalUsages1 = new ulong[links.Count() + 1];
978
979                     sequences.CalculateAllUsages(linksTotalUsages1);
980
981                     var linksTotalUsages2 = new ulong[links.Count() + 1];
982
983                     sequences.CalculateAllUsages2(linksTotalUsages2);
```

```
984
985                     var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
986                     Assert.True(intersection1.Count == linksTotalUsages2.Length);
987                 }
988
989                 for (var i = 0; i < sequenceLength; i++)
990                 {
991                     links.Delete(sequence[i]);
992                 }
993             }
994         }
995     }
996 }
```

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

```
1  using System.IO;
2  using Platform.Disposables;
3  using Platform.Data.Doublets.ResizableDirectMemory;
4  using Platform.Data.Doublets.Sequences;
5  using Platform.Data.Doublets.Decorators;
6
7  namespace Platform.Data.Doublets.Tests
8  {
9      public class TempLinksTestScope : DisposableBase
10     {
11         public readonly ILinks<ulong> MemoryAdapter;
12         public readonly SynchronizedLinks<ulong> Links;
13         public readonly Sequences.Sequences Sequences;
14         public readonly string TempFilename;
15         public readonly string TempTransactionLogFilename;
16         private readonly bool _deleteFiles;
17
18         public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool
           ↪  useLog = false)
19             : this(new SequencesOptions<ulong>(), deleteFiles, useSequences, useLog)
20         {
21         }
22
23         public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles =
           ↪  true, bool useSequences = false, bool useLog = false)
24         {
25             _deleteFiles = deleteFiles;
26             TempFilename = Path.GetTempFileName();
27             TempTransactionLogFilename = Path.GetTempFileName();
28
29             var coreMemoryAdapter = new UInt64ResizableDirectMemoryLinks(TempFilename);
30
31             MemoryAdapter = useLog ? (ILinks<ulong>)new
               ↪  UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
               ↪  coreMemoryAdapter;
32
33             Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
34             if (useSequences)
35             {
36                 Sequences = new Sequences.Sequences(Links, sequencesOptions);
37             }
38         }
39
40         protected override void Dispose(bool manual, bool wasDisposed)
41         {
42             if (!wasDisposed)
43             {
44                 Links.Unsync.DisposeIfPossible();
45                 if (_deleteFiles)
46                 {
47                     DeleteFiles();
48                 }
49             }
50         }
51
52         public void DeleteFiles()
53         {
54             File.Delete(TempFilename);
55             File.Delete(TempTransactionLogFilename);
56         }
57     }
58 }
```

Index