LinksPlatform's Platform.Data.Doublets Class Library

1.1 ./csharp/Platform.Data.Doublets/CriterionMatchers/TargetMatcher.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.CriterionMatchers
{
    public class TargetMatcher<TLink> : LinksOperatorBase<TLink>, ICriterionMatcher<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly TLink _targetToMatch;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TargetMatcher(ILinks<TLink> links, TLink targetToMatch) : base(links) =>
        ↪  _targetToMatch = targetToMatch;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool IsMatched(TLink link) => _equalityComparer.Equals(_links.GetTarget(link),
        ↪  _targetToMatch);
    }
}
```

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

```csharp
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Decorators
{
    public class LinksCascadeUniquenessAndUsagesResolver<TLink> : LinksUniquenessResolver<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksCascadeUniquenessAndUsagesResolver(ILinks<TLink> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
        ↪  newLinkAddress)
        {
            // Use Facade (the last decorator) to ensure recursion working correctly
            _facade.MergeUsages(oldLinkAddress, newLinkAddress);
            return base.ResolveAddressChangeConflict(oldLinkAddress, newLinkAddress);
        }
    }
}
```

1.3 ./csharp/Platform.Data.Doublets/Decorators/LinksCascadeUsagesResolver.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Decorators
{
    /// <remarks>
    /// <para>Must be used in conjunction with NonNullContentsLinkDeletionResolver.</para>
    /// <para>Должен использоваться вместе с NonNullContentsLinkDeletionResolver.</para>
    /// </remarks>
    public class LinksCascadeUsagesResolver<TLink> : LinksDecoratorBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksCascadeUsagesResolver(ILinks<TLink> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override void Delete(IList<TLink> restrictions)
        {
            var linkIndex = restrictions[_constants.IndexPart];
            // Use Facade (the last decorator) to ensure recursion working correctly
            _facade.DeleteAllUsages(linkIndex);
            _links.Delete(linkIndex);
        }
    }
}
```

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Decorators
{
    public abstract class LinksDecoratorBase<TLink> : LinksOperatorBase<TLink>, ILinks<TLink>
    {
        protected readonly LinksConstants<TLink> _constants;

        public LinksConstants<TLink> Constants
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _constants;
        }

        protected ILinks<TLink> _facade;

        public ILinks<TLink> Facade
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _facade;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set
            {
                _facade = value;
                if (_links is LinksDecoratorBase<TLink> decorator)
                {
                    decorator.Facade = value;
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected LinksDecoratorBase(ILinks<TLink> links) : base(links)
        {
            _constants = links.Constants;
            Facade = this;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Count(IList<TLink> restrictions) => _links.Count(restrictions);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
        ↪  => _links.Each(handler, restrictions);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Create(IList<TLink> restrictions) => _links.Create(restrictions);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
        ↪  _links.Update(restrictions, substitution);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual void Delete(IList<TLink> restrictions) => _links.Delete(restrictions);
    }
}
```

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

```csharp
using System.Runtime.CompilerServices;
using Platform.Disposables;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
#pragma warning disable CA1063 // Implement IDisposable Correctly

namespace Platform.Data.Doublets.Decorators
{
    public abstract class LinksDisposableDecoratorBase<TLink> : LinksDecoratorBase<TLink>,
    ↪  ILinks<TLink>, System.IDisposable
    {
        protected class DisposableWithMultipleCallsAllowed : Disposable
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public DisposableWithMultipleCallsAllowed(Disposal disposal) : base(disposal) { }

            protected override bool AllowMultipleDisposeCalls
            {
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                get => true;
            }
        }

        protected readonly DisposableWithMultipleCallsAllowed Disposable;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected LinksDisposableDecoratorBase(ILinks<TLink> links) : base(links) => Disposable
         ↪  = new DisposableWithMultipleCallsAllowed(Dispose);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        ~LinksDisposableDecoratorBase() => Disposable.Destruct();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Dispose() => Disposable.Dispose();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                _links.DisposeIfPossible();
            }
        }
    }
}
```

1.6 ./csharp/Platform.Data.Doublets/Decorators/LinksInnerReferenceExistenceValidator.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Decorators
{
    // TODO: Make LinksExternalReferenceValidator. A layer that checks each link to exist or to
     ↪  be external (hybrid link's raw number).
    public class LinksInnerReferenceExistenceValidator<TLink> : LinksDecoratorBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksInnerReferenceExistenceValidator(ILinks<TLink> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
        {
            var links = _links;
            links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
            return links.Each(handler, restrictions);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
        {
            // TODO: Possible values: null, ExistentLink or NonExistentHybrid(ExternalReference)
            var links = _links;
            links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
            links.EnsureInnerReferenceExists(substitution, nameof(substitution));
            return links.Update(restrictions, substitution);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override void Delete(IList<TLink> restrictions)
        {
            var link = restrictions[_constants.IndexPart];
            var links = _links;
            links.EnsureLinkExists(link, nameof(link));
            links.Delete(link);
        }
    }
}
```

1.7 ./csharp/Platform.Data.Doublets/Decorators/LinksItselfConstantToSelfReferenceResolver.cs

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

```

```csharp
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksItselfConstantToSelfReferenceResolver(ILinks<TLink> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
        {
            var constants = _constants;
            var itselfConstant = constants.Itself;
            if (!_equalityComparer.Equals(constants.Any, itselfConstant) &&
            ↪  restrictions.Contains(itselfConstant))
            {
                // Itself constant is not supported for Each method right now, skipping execution
                return constants.Continue;
            }
            return _links.Each(handler, restrictions);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
        ↪  _links.Update(restrictions, _links.ResolveConstantAsSelfReference(_constants.Itself,
        ↪  restrictions, substitution));
    }
}
```

1.8 ./csharp/Platform.Data.Doublets/Decorators/LinksNonExistentDependenciesCreator.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Decorators
{
    /// <remarks>
    /// Not practical if newSource and newTarget are too big.
    /// To be able to use practical version we should allow to create link at any specific
    ↪  location inside ResizableDirectMemoryLinks.
    /// This in turn will require to implement not a list of empty links, but a list of ranges
    ↪  to store it more efficiently.
    /// </remarks>
    public class LinksNonExistentDependenciesCreator<TLink> : LinksDecoratorBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksNonExistentDependenciesCreator(ILinks<TLink> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
        {
            var constants = _constants;
            var links = _links;
            links.EnsureCreated(substitution[constants.SourcePart],
            ↪  substitution[constants.TargetPart]);
            return links.Update(restrictions, substitution);
        }
    }
}
```

1.9 ./csharp/Platform.Data.Doublets/Decorators/LinksNullConstantToSelfReferenceResolver.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Decorators
{
    public class LinksNullConstantToSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksNullConstantToSelfReferenceResolver(ILinks<TLink> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Create(IList<TLink> restrictions) => _links.CreatePoint();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
        ↪  _links.Update(restrictions, _links.ResolveConstantAsSelfReference(_constants.Null,
        ↪  restrictions, substitution));
    }
}
```

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksUniquenessResolver(ILinks<TLink> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
        {
            var constants = _constants;
            var links = _links;
            var newLinkAddress = links.SearchOrDefault(substitution[constants.SourcePart],
            ↪  substitution[constants.TargetPart]);
            if (_equalityComparer.Equals(newLinkAddress, default))
            {
                return links.Update(restrictions, substitution);
            }
            return ResolveAddressChangeConflict(restrictions[constants.IndexPart],
            ↪  newLinkAddress);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
        ↪  newLinkAddress)
        {
            if (!_equalityComparer.Equals(oldLinkAddress, newLinkAddress) &&
            ↪  _links.Exists(oldLinkAddress))
            {
                _facade.Delete(oldLinkAddress);
            }
            return newLinkAddress;
        }
    }
}
```

1.11 ./csharp/Platform.Data.Doublets/Decorators/LinksUniquenessValidator.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Decorators
{
    public class LinksUniquenessValidator<TLink> : LinksDecoratorBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksUniquenessValidator(ILinks<TLink> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
        {
            var links = _links;
            var constants = _constants;
            links.EnsureDoesNotExists(substitution[constants.SourcePart],
            ↪  substitution[constants.TargetPart]);
            return links.Update(restrictions, substitution);
        }
    }
}
```

1.12 ./csharp/Platform.Data.Doublets/Decorators/LinksUsagesValidator.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Decorators
{
    public class LinksUsagesValidator<TLink> : LinksDecoratorBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksUsagesValidator(ILinks<TLink> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
        {
            var links = _links;
            links.EnsureNoUsages(restrictions[_constants.IndexPart]);
            return links.Update(restrictions, substitution);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override void Delete(IList<TLink> restrictions)
        {
            var link = restrictions[_constants.IndexPart];
            var links = _links;
            links.EnsureNoUsages(link);
            links.Delete(link);
        }
    }
}
```

1.13 ./csharp/Platform.Data.Doublets/Decorators/NonNullContentsLinkDeletionResolver.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Decorators
{
    public class NonNullContentsLinkDeletionResolver<TLink> : LinksDecoratorBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public NonNullContentsLinkDeletionResolver(ILinks<TLink> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override void Delete(IList<TLink> restrictions)
        {
            var linkIndex = restrictions[_constants.IndexPart];
            var links = _links;
            links.EnforceResetValues(linkIndex);
            links.Delete(linkIndex);
        }
    }
}
```

1.14 ./csharp/Platform.Data.Doublets/Decorators/UInt64Links.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Decorators
{
    /// <summary>
    /// <para>Represents a combined decorator that implements the basic logic for interacting
    ↪  with the links storage for links with addresses represented as <see cref="System.UInt64"
    ↪  />.</para>
    /// <para>Представляет комбинированный декоратор, реализующий основную логику по
    ↪  взаимодействии с хранилищем связей, для связей с адресами представленными в виде <see
    ↪  cref="System.UInt64"/>.</para>
    /// </summary>
    /// <remarks>
    /// Возможные оптимизации:
    /// Объединение в одном поле Source и Target с уменьшением до 32 бит.
    ///     + меньше объём БД
    ///     - меньше производительность
    ///     - больше ограничение на количество связей в БД)
    /// Ленивое хранение размеров поддеревьев (расчитываемое по мере использования БД)
    ///     + меньше объём БД
    ///     - больше сложность
    ///
    /// Текущее теоретическое ограничение на индекс связи, из-за использования 5 бит в размере
    ↪  поддеревьев для AVL баланса и флагов нитей: 2 в степени(64 минус 5 равно 59 ) равно 576
    ↪  460 752 303 423 488
    /// Желательно реализовать поддержку переключения между деревьями и битовыми индексами
    ↪  (битовыми строками) - вариант матрицы (выстраеваемой лениво).
    ///
    /// Решить отключать ли проверки при компиляции под Release. Т.е. исключения будут
    ↪  выбрасываться только при #if DEBUG
    /// </remarks>
    public class UInt64Links : LinksDisposableDecoratorBase<ulong>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64Links(ILinks<ulong> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override ulong Create(IList<ulong> restrictions) => _links.CreatePoint();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
        {
            var constants = _constants;
            var indexPartConstant = constants.IndexPart;
            var sourcePartConstant = constants.SourcePart;
            var targetPartConstant = constants.TargetPart;
            var nullConstant = constants.Null;
            var itselfConstant = constants.Itself;
            var existedLink = nullConstant;
            var updatedLink = restrictions[indexPartConstant];
            var newSource = substitution[sourcePartConstant];
            var newTarget = substitution[targetPartConstant];
            var links = _links;
            if (newSource != itselfConstant && newTarget != itselfConstant)
            {
                existedLink = links.SearchOrDefault(newSource, newTarget);
            }
            if (existedLink == nullConstant)
            {
                var before = links.GetLink(updatedLink);
                if (before[sourcePartConstant] != newSource || before[targetPartConstant] !=
                ↪  newTarget)
                {
                    links.Update(updatedLink, newSource == itselfConstant ? updatedLink :
                    ↪  newSource,
                                                  newTarget == itselfConstant ? updatedLink :
                                                  ↪  newTarget);
                }
                return updatedLink;
            }
            else
            {
                return _facade.MergeAndDelete(updatedLink, existedLink);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override void Delete(IList<ulong> restrictions)
        {
            var linkIndex = restrictions[_constants.IndexPart];
            var links = _links;
            links.EnforceResetValues(linkIndex);
            _facade.DeleteAllUsages(linkIndex);
            links.Delete(linkIndex);
        }
    }
}
```

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

```

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

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

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

```
            //        matchedLinks = new List<IList<T>>();
            //        Func<T, bool> handler = link =>
            //        {
            //            var matchedLink = Memory.GetLinkValue(link);
            //            var matchDecision = matchedHandler(matchedLink);
            //            if (Equals(matchDecision, Constants.Break))
            //                return false;
            //            if (!Equals(matchDecision, Constants.Skip))
            //                matchedLinks.Add(matchedLink);
            //            return true;
            //        };
            //        if (!Memory.Each(handler, restriction))
            //            return Constants.Break;
            //    }
            //    if (!matchedLinks.IsNullOrEmpty())
            //    {
            //        var totalMatchedLinks = matchedLinks.Count;
            //        for (var i = 0; i < totalMatchedLinks; i++)
            //        {
            //            var matchedLink = matchedLinks[i];
            //            if (substitutedHandler != null)
            //            {
            //                var newValue = new List<T>(); // TODO: Prepare value to update here
            //                // TODO: Decide is it actually needed to use Before and After substitution handling.
            //                var substitutedDecision = substitutedHandler(matchedLink, newValue);
            //                if (Equals(substitutedDecision, Constants.Break))
            //                    return Constants.Break;
            //                if (Equals(substitutedDecision, Constants.Continue))
            //                {
            //                    // Actual update here
            //                    Memory.SetLinkValue(newValue);
            //                }
            //                if (Equals(substitutedDecision, Constants.Skip))
            //                {
            //                    // Cancel the update. TODO: decide use separate Cancel constant or Skip is enough?
            //                }
            //            }
            //        }
            //    }
            //}
            return _constants.Continue;
        }

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

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

1.16 ./csharp/Platform.Data.Doublets/Doublet.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

        public T Source
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }
        public T Target
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Doublet(T source, T target)
        {
            Source = source;
            Target = target;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override string ToString() => $"{Source}->{Target}";

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override int GetHashCode() => (Source, Target).GetHashCode();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator ==(Doublet<T> left, Doublet<T> right) => left.Equals(right);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator !=(Doublet<T> left, Doublet<T> right) => !(left == right);
    }
}
```

1.17 ./csharp/Platform.Data.Doublets/DoubletComparer.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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();
    }
}
```

1.18 ./csharp/Platform.Data.Doublets/ILinks.cs

```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

using System.Collections.Generic;

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

1.19 ./csharp/Platform.Data.Doublets/ILinksExtensions.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.CompilerServices;
using Platform.Ranges;
using Platform.Collections.Arrays;
using Platform.Random;
using Platform.Setters;
using Platform.Converters;
using Platform.Numbers;
using Platform.Data.Exceptions;
using Platform.Data.Doublets.Decorators;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets
{
    public static class ILinksExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void RunRandomCreations<TLink>(this ILinks<TLink> links, ulong
         ↪  amountOfCreations)
        {
            var random = RandomHelpers.Default;
            var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
            var uInt64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
            for (var i = 0UL; i < amountOfCreations; i++)
            {
                var linksAddressRange = new Range<ulong>(0,
                ↪   addressToUInt64Converter.Convert(links.Count()));
                var source =
                ↪   uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
                var target =
                ↪   uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
                links.GetOrCreate(source, target);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void RunRandomSearches<TLink>(this ILinks<TLink> links, ulong
         ↪  amountOfSearches)
        {
            var random = RandomHelpers.Default;
            var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
            var uInt64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
            for (var i = 0UL; i < amountOfSearches; i++)
            {
                var linksAddressRange = new Range<ulong>(0,
                ↪   addressToUInt64Converter.Convert(links.Count()));
                var source =
                ↪   uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
                var target =
                ↪   uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
                links.SearchOrDefault(source, target);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void RunRandomDeletions<TLink>(this ILinks<TLink> links, ulong
         ↪  amountOfDeletions)
        {
            var random = RandomHelpers.Default;
            var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
            var uInt64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
            var linksCount = addressToUInt64Converter.Convert(links.Count());
            var min = amountOfDeletions > linksCount ? 0UL : linksCount - amountOfDeletions;
            for (var i = 0UL; i < amountOfDeletions; i++)
            {
                linksCount = addressToUInt64Converter.Convert(links.Count());
                if (linksCount <= min)
                {
                    break;
                }
                var linksAddressRange = new Range<ulong>(min, linksCount);
                var link =
                ↪   uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
                links.Delete(link);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Delete<TLink>(this ILinks<TLink> links, TLink linkToDelete) =>
         ↪  links.Delete(new LinkAddress<TLink>(linkToDelete));

        /// <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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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();
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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 InvalidOperationException("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 InvalidOperationException("В процессе поиска по хранилищу не было
                ↪  найдено связей.");
            }
            return firstLink;
        }

        #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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[]
        ↪  path)
        {
            var current = path[0];
            //EnsureLinkExists(current, "path");
            if (!links.Exists(current))
            {
                return false;
            }
            var equalityComparer = EqualityComparer<TLink>.Default;
            var constants = links.Constants;
            for (var i = 1; i < path.Length; i++)
            {
                var next = path[i];
                var values = links.GetLink(current);
                var source = values[constants.SourcePart];
                var target = values[constants.TargetPart];
                if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
                ↪  next))
                {
                    //throw new InvalidOperationException(string.Format("Невозможно выбрать
                    ↪  путь, так как и Source и Target совпадают с элементом пути {0}.", next));
                    return false;
                }
                if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
                ↪  target))
                {
                    //throw new InvalidOperationException(string.Format("Невозможно продолжить
                    ↪  путь через элемент пути {0}", next));
                    return false;
                }
                current = next;
            }
            return true;
        }

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

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

        #endregion

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

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

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

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

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

        /// <summary>
        /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
         ↪  (handler) для каждой подходящей связи.
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="handler">Обработчик каждой подходящей связи.</param>
        /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
         ↪  может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
         ↪  Any - отсутствие ограничения, 1..∞ конкретный адрес связи.</param>
        /// <returns>True, в случае если проход по связям не был прерван и False в обратном
         ↪  случае.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
         ↪  handler, params TLink[] restrictions)
            => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
             ↪  links.Constants.Continue);

        /// <summary>
        /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
         ↪  (handler) для каждой подходящей связи.
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="source">Значение, определяющее соответствующие шаблону связи.
         ↪  (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
         ↪  Constants.Any - любое начало, 1..∞ конкретное начало)</param>
        /// <param name="target">Значение, определяющее соответствующие шаблону связи.
         ↪  (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
         ↪  Constants.Any - любой конец, 1..∞ конкретный конец)</param>
        /// <param name="handler">Обработчик каждой подходящей связи.</param>
        /// <returns>True, в случае если проход по связям не был прерван и False в обратном
         ↪  случае.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
         ↪  Func<TLink, bool> handler)
        {
            var constants = links.Constants;
            return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
             ↪  constants.Break, constants.Any, source, target);
        }

        /// <summary>
        /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
         ↪  (handler) для каждой подходящей связи.
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="source">Значение, определяющее соответствующие шаблону связи.
         ↪  (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
         ↪  Constants.Any - любое начало, 1..∞ конкретное начало)</param>
        /// <param name="target">Значение, определяющее соответствующие шаблону связи.
         ↪  (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
         ↪  Constants.Any - любой конец, 1..∞ конкретный конец)</param>
        /// <param name="handler">Обработчик каждой подходящей связи.</param>
        /// <returns>True, в случае если проход по связям не был прерван и False в обратном
         ↪  случае.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
         ↪  Func<IList<TLink>, TLink> handler) => links.Each(handler, links.Constants.Any,
         ↪  source, target);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
         ↪  restrictions)
        {
            var arraySize = CheckedConverter<TLink,
             ↪  long>.Default.Convert(links.Count(restrictions));
            if (arraySize > 0)
            {
                var array = new IList<TLink>[arraySize];
                var filler = new ArrayFiller<IList<TLink>, TLink>(array,
                 ↪  links.Constants.Continue);
                links.Each(filler.AddAndReturnConstant, restrictions);
                return array;
            }
            else
            {
                return Array.Empty<IList<TLink>>();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IList<TLink> AllIndices<TLink>(this ILinks<TLink> links, params TLink[]
         ↪  restrictions)
        {
            var arraySize = CheckedConverter<TLink,
             ↪  long>.Default.Convert(links.Count(restrictions));
            if (arraySize > 0)
            {
                var array = new TLink[arraySize];
                var filler = new ArrayFiller<TLink, TLink>(array, links.Constants.Continue);
                links.Each(filler.AddFirstAndReturnConstant, restrictions);
                return array;
            }
            else
            {
                return Array.Empty<TLink>();
            }
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureLinkExists<TLink>(this ILinks<TLink> links, IList<TLink>
         ↪  restrictions)
        {
            for (var i = 0; i < restrictions.Count; i++)
            {
                if (!links.Exists(restrictions[i]))
                {
                    throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
                     ↪  $"sequence[{i}]");
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
         ↪  reference, string argumentName)
        {
            if (links.Constants.IsInternalReference(reference) && !links.Exists(reference))
            {
                throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
            }
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
        ↪  restrictions)
        {
            var equalityComparer = EqualityComparer<TLink>.Default;
            var any = links.Constants.Any;
            for (var i = 0; i < restrictions.Count; i++)
            {
                if (!equalityComparer.Equals(restrictions[i], any) &&
                ↪  !links.Exists(restrictions[i]))
                {
                    throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
                    ↪  $"sequence[{i}]");
                }
            }
        }

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

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

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

        /// <param name="links">Хранилище связей.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureNoUsages<TLink>(this ILinks<TLink> links, TLink link)
        {
            if (links.HasUsages(link))
            {
                throw new ArgumentLinkHasDependenciesException<TLink>(link);
            }
        }

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

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

        /// <param name="links">Хранилище связей.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
        ↪  params TLink[] addresses)
        {
            var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
            var uInt64ToAddressConverter = CheckedConverter<ulong, TLink>.Default;
            var nonExistentAddresses = new HashSet<TLink>(addresses.Where(x => !links.Exists(x)));
            if (nonExistentAddresses.Count > 0)
            {
                var max = nonExistentAddresses.Max();
                max = uInt64ToAddressConverter.Convert(System.Math.Min(addressToUInt64Converter.Convert(max), addressToUInt64Converter.Convert(links.Constants.InternalReferencesRange.Maximum)));
                var createdLinks = new List<TLink>();
                var equalityComparer = EqualityComparer<TLink>.Default;
                TLink createdLink = creator();
                while (!equalityComparer.Equals(createdLink, max))
                {
                    createdLinks.Add(createdLink);
                }
                for (var i = 0; i < createdLinks.Count; i++)
                {
                    if (!nonExistentAddresses.Contains(createdLinks[i]))
                    {
                        links.Delete(createdLinks[i]);
                    }
                }
            }
        }

        #endregion

        /// <param name="links">Хранилище связей.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLink CountUsages<TLink>(this ILinks<TLink> links, TLink link)
        {
            var constants = links.Constants;
            var values = links.GetLink(link);
            TLink usagesAsSource = links.Count(new Link<TLink>(constants.Any, link, constants.Any));
            var equalityComparer = EqualityComparer<TLink>.Default;
            if (equalityComparer.Equals(values[constants.SourcePart], link))
            {
                usagesAsSource = Arithmetic<TLink>.Decrement(usagesAsSource);
            }
            TLink usagesAsTarget = links.Count(new Link<TLink>(constants.Any, constants.Any, link));
            if (equalityComparer.Equals(values[constants.TargetPart], link))
            {
                usagesAsTarget = Arithmetic<TLink>.Decrement(usagesAsTarget);
            }
            return Arithmetic<TLink>.Add(usagesAsSource, usagesAsTarget);
        }

        /// <param name="links">Хранилище связей.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool HasUsages<TLink>(this ILinks<TLink> links, TLink link) => Comparer<TLink>.Default.Compare(links.CountUsages(link), default) > 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>
```

```csharp
        /// <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 Create<TLink>(this ILinks<TLink> links) => links.Create(null);

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
        ↪  links, TLink constant, IList<TLink> restrictions, IList<TLink> substitution)
        {
            var equalityComparer = EqualityComparer<TLink>.Default;
            var constants = links.Constants;
            var restrictionsIndex = restrictions[constants.IndexPart];
            var substitutionIndex = substitution[constants.IndexPart];
            if (equalityComparer.Equals(substitutionIndex, default))
            {
                substitutionIndex = restrictionsIndex;
            }
            var source = substitution[constants.SourcePart];
            var target = substitution[constants.TargetPart];
            source = equalityComparer.Equals(source, constant) ? substitutionIndex : source;
            target = equalityComparer.Equals(target, constant) ? substitutionIndex : target;
            return new Link<TLink>(substitutionIndex, source, target);
        }

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

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

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

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

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

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

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

        // TODO: Create a universal version of this method in Platform.Data (with using of for
        ↪  loop)
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void ResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
        {
            var nullConstant = links.Constants.Null;
            var updateRequest = new Link<TLink>(linkIndex, nullConstant, nullConstant);
            links.Update(updateRequest);
        }

        // TODO: Create a universal version of this method in Platform.Data (with using of for
        ↪  loop)
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnforceResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
        {
            if (!links.AreValuesReset(linkIndex))
            {
                links.ResetValues(linkIndex);
            }
        }

        /// <summary>
        /// Merging two usages graphs, all children of old link moved to be children of new link
        ↪  or deleted.
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLink MergeUsages<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
        ↪  TLink newLinkIndex)
        {
            var addressToInt64Converter = CheckedConverter<TLink, long>.Default;
            var equalityComparer = EqualityComparer<TLink>.Default;
            if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
            {
                var constants = links.Constants;
                var usagesAsSourceQuery = new Link<TLink>(constants.Any, oldLinkIndex,
                ↪  constants.Any);
                var usagesAsSourceCount =
                ↪  addressToInt64Converter.Convert(links.Count(usagesAsSourceQuery));
                var usagesAsTargetQuery = new Link<TLink>(constants.Any, constants.Any,
                ↪  oldLinkIndex);
                var usagesAsTargetCount =
                ↪  addressToInt64Converter.Convert(links.Count(usagesAsTargetQuery));
                var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(oldLinkIndex)) &&
                ↪  usagesAsSourceCount == 1 && usagesAsTargetCount == 1;
                if (!isStandalonePoint)
                {
                    var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
                    if (totalUsages > 0)
                    {
                        var usages = ArrayPool.Allocate<TLink>(totalUsages);
                        var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
                        ↪  links.Constants.Continue);
                        var i = 0L;
                        if (usagesAsSourceCount > 0)
                        {
                            links.Each(usagesFiller.AddFirstAndReturnConstant,
                            ↪  usagesAsSourceQuery);
                            for (; i < usagesAsSourceCount; i++)
                            {
                                var usage = usages[i];
                                if (!equalityComparer.Equals(usage, oldLinkIndex))
                                {
                                    links.Update(usage, newLinkIndex, links.GetTarget(usage));
                                }
                            }
                        }
                        if (usagesAsTargetCount > 0)
                        {
                            links.Each(usagesFiller.AddFirstAndReturnConstant,
                            ↪  usagesAsTargetQuery);
                            for (; i < usages.Length; i++)
                            {
                                var usage = usages[i];
                                if (!equalityComparer.Equals(usage, oldLinkIndex))
                                {
                                    links.Update(usage, links.GetSource(usage), newLinkIndex);
                                }
                            }
                        }
                        ArrayPool.Free(usages);
                    }
                }
            }
            return newLinkIndex;
        }

        /// <summary>
        /// Replace one link with another (replaced link is deleted, children are updated or
        ↪  deleted).
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
        ↪  TLink newLinkIndex)
        {
            var equalityComparer = EqualityComparer<TLink>.Default;
            if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
            {
                links.MergeUsages(oldLinkIndex, newLinkIndex);
                links.Delete(oldLinkIndex);
            }
            return newLinkIndex;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ILinks<TLink>
        ↪  DecorateWithAutomaticUniquenessAndUsagesResolution<TLink>(this ILinks<TLink> links)
        {
            links = new LinksCascadeUsagesResolver<TLink>(links);
            links = new NonNullContentsLinkDeletionResolver<TLink>(links);
            links = new LinksCascadeUniquenessAndUsagesResolver<TLink>(links);
            return links;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string Format<TLink>(this ILinks<TLink> links, IList<TLink> link)
        {
            var constants = links.Constants;
            return $"({link[constants.IndexPart]}: {link[constants.SourcePart]}
            ↪  {link[constants.TargetPart]})";
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string Format<TLink>(this ILinks<TLink> links, TLink link) =>
        ↪  links.Format(links.GetLink(link));
    }
}
```

1.20 ./csharp/Platform.Data.Doublets/ISynchronizedLinks.cs

```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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;

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Increment(TLink frequency)
        {
```

```csharp
            var links = _links;
            if (_equalityComparer.Equals(frequency, default))
            {
                return links.GetOrCreate(_unaryOne, _frequencyMarker);
            }
            var incrementedSource =
             ↪  _unaryNumberIncrementer.Increment(links.GetSource(frequency));
            return links.GetOrCreate(incrementedSource, _frequencyMarker);
        }
    }
}
```

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

```csharp
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Incrementers;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) : base(links) =>
         ↪  _unaryOne = unaryOne;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Increment(TLink unaryNumber)
        {
            var links = _links;
            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));
            }
        }
    }
}
```

1.23 ./csharp/Platform.Data.Doublets/Link.cs

```csharp
using Platform.Collections.Lists;
using Platform.Exceptions;
using Platform.Ranges;
using Platform.Singletons;
using System;
using System.Collections;
using System.Collections.Generic;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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 LinksConstants<TLink> _constants =
         ↪  Default<LinksConstants<TLink>>.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;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Link(params TLink[] values) => SetValues(values, out Index, out Source, out
         ↪  Target);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Link(IList<TLink> values) => SetValues(values, out Index, out Source, out Target);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Link(object other)
        {
            if (other is Link<TLink> otherLink)
            {
                SetValues(ref otherLink, out Index, out Source, out Target);
            }
            else if(other is IList<TLink> otherList)
            {
                SetValues(otherList, out Index, out Source, out Target);
            }
            else
            {
                throw new NotSupportedException();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Link(ref Link<TLink> other) => SetValues(ref other, out Index, out Source, out
         ↪  Target);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Link(TLink index, TLink source, TLink target)
        {
            Index = index;
            Source = source;
            Target = target;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void SetValues(ref Link<TLink> other, out TLink index, out TLink source,
         ↪  out TLink target)
        {
            index = other.Index;
            source = other.Source;
            target = other.Target;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void SetValues(IList<TLink> values, out TLink index, out TLink source,
         ↪  out TLink target)
        {
            switch (values.Count)
            {
                case 3:
                    index = values[0];
                    source = values[1];
                    target = values[2];
                    break;
                case 2:
                    index = values[0];
                    source = values[1];
                    target = default;
                    break;
                case 1:
                    index = values[0];
                    source = default;
                    target = default;
                    break;
                default:
                    index = default;
                    source = default;
                    target = default;
                    break;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override int GetHashCode() => (Index, Source, Target).GetHashCode();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool IsNull() => _equalityComparer.Equals(Index, _constants.Null)
                             && _equalityComparer.Equals(Source, _constants.Null)
                             && _equalityComparer.Equals(Target, _constants.Null);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Equals(object other) => other is Link<TLink> &&
        ↪  Equals((Link<TLink>)other);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
                                              && _equalityComparer.Equals(Source, other.Source)
                                              && _equalityComparer.Equals(Target, other.Target);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string ToString(TLink index, TLink source, TLink target) => $"({index}:
        ↪  {source}->{target})";

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string ToString(TLink source, TLink target) => $"({source}->{target})";

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

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

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

        #region IList

        public int Count
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => Length;
        }

        public bool IsReadOnly
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => true;
        }

        public TLink this[int index]
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get
            {
                Ensure.OnDebug.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
            }
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set => throw new NotSupportedException();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IEnumerator<TLink> GetEnumerator()
        {
            yield return Index;
            yield return Source;
            yield return Target;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Add(TLink item) => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Clear() => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Contains(TLink item) => IndexOf(item) >= 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void CopyTo(TLink[] array, int arrayIndex)
        {
            Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
            Ensure.OnDebug.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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Remove(TLink item) => Throw.A.NotSupportedExceptionAndReturn<bool>();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Insert(int index, TLink item) => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void RemoveAt(int index) => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator ==(Link<TLink> left, Link<TLink> right) =>
         ↪  left.Equals(right);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator !=(Link<TLink> left, Link<TLink> right) => !(left == right);

        #endregion
    }
}
```

1.24 ./csharp/Platform.Data.Doublets/LinkExtensions.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets
{
    public static class LinkExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsFullPoint<TLink>(this Link<TLink> link) =>
         ↪  Point<TLink>.IsFullPoint(link);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsPartialPoint<TLink>(this Link<TLink> link) =>
         ↪  Point<TLink>.IsPartialPoint(link);
    }
}
```

1.25 ./csharp/Platform.Data.Doublets/LinksOperatorBase.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets
{
    public abstract class LinksOperatorBase<TLink>
    {
        protected readonly ILinks<TLink> _links;

        public ILinks<TLink> Links
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _links;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected LinksOperatorBase(ILinks<TLink> links) => _links = links;
    }
}
```

1.26 ./csharp/Platform.Data.Doublets/Memory/ILinksListMethods.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory
{
    public interface ILinksListMethods<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        void Detach(TLink freeLink);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        void AttachAsFirst(TLink link);
    }
}
```

1.27 ./csharp/Platform.Data.Doublets/Memory/ILinksTreeMethods.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory
{
    public interface ILinksTreeMethods<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        TLink CountUsages(TLink root);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        TLink Search(TLink source, TLink target);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        TLink EachUsage(TLink root, Func<IList<TLink>, TLink> handler);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        void Detach(ref TLink root, TLink linkIndex);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        void Attach(ref TLink root, TLink linkIndex);
    }
}
```

1.28 ./csharp/Platform.Data.Doublets/Memory/LinksHeader.cs

```csharp
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Unsafe;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

        public static readonly long SizeInBytes = Structure<LinksHeader<TLink>>.Size;

        public TLink AllocatedLinks;
        public TLink ReservedLinks;
        public TLink FreeLinks;
        public TLink FirstFreeLink;
        public TLink RootAsSource;
        public TLink RootAsTarget;
        public TLink LastFreeLink;
        public TLink Reserved8;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Equals(object obj) => obj is LinksHeader<TLink> linksHeader ?
         ↪  Equals(linksHeader) : false;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Equals(LinksHeader<TLink> other)
            => _equalityComparer.Equals(AllocatedLinks, other.AllocatedLinks)
            && _equalityComparer.Equals(ReservedLinks, other.ReservedLinks)
            && _equalityComparer.Equals(FreeLinks, other.FreeLinks)
            && _equalityComparer.Equals(FirstFreeLink, other.FirstFreeLink)
            && _equalityComparer.Equals(RootAsSource, other.RootAsSource)
            && _equalityComparer.Equals(RootAsTarget, other.RootAsTarget)
            && _equalityComparer.Equals(LastFreeLink, other.LastFreeLink)
            && _equalityComparer.Equals(Reserved8, other.Reserved8);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override int GetHashCode() => (AllocatedLinks, ReservedLinks, FreeLinks,
         ↪  FirstFreeLink, RootAsSource, RootAsTarget, LastFreeLink, Reserved8).GetHashCode();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator ==(LinksHeader<TLink> left, LinksHeader<TLink> right) =>
         ↪  left.Equals(right);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator !=(LinksHeader<TLink> left, LinksHeader<TLink> right) =>
         ↪  !(left == right);
    }
}
```

1.29 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSizeBalancedTreeMethodsBase.cs

```csharp
using System;
using System.Text;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections.Methods.Trees;
using Platform.Converters;
using static System.Runtime.CompilerServices.Unsafe;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.Split.Generic
{
    public unsafe abstract class ExternalLinksSizeBalancedTreeMethodsBase<TLink> :
     ↪  SizeBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
    {
        private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
         ↪  UncheckedConverter<TLink, long>.Default;

        protected readonly TLink Break;
        protected readonly TLink Continue;
        protected readonly byte* LinksDataParts;
        protected readonly byte* LinksIndexParts;
        protected readonly byte* Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected ExternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants,
         ↪  byte* linksDataParts, byte* linksIndexParts, byte* header)
        {
            LinksDataParts = linksDataParts;
            LinksIndexParts = linksIndexParts;
            Header = header;
            Break = constants.Break;
            Continue = constants.Continue;
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
         ↪  rootSource, TLink rootTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
         ↪  rootSource, TLink rootTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
         ↪  AsRef<LinksHeader<TLink>>(Header);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
         ↪  AsRef<RawLinkDataPart<TLink>>(LinksDataParts + RawLinkDataPart<TLink>.SizeInBytes *
         ↪  _addressToInt64Converter.Convert(link));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
         ↪  ref AsRef<RawLinkIndexPart<TLink>>(LinksIndexParts +
         ↪  RawLinkIndexPart<TLink>.SizeInBytes * _addressToInt64Converter.Convert(link));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
        {
            ref var link = ref GetLinkDataPartReference(linkIndex);
            return new Link<TLink>(linkIndex, link.Source, link.Target);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
        {
            ref var firstLink = ref GetLinkDataPartReference(first);
            ref var secondLink = ref GetLinkDataPartReference(second);
            return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
             ↪  secondLink.Source, secondLink.Target);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
        {
            ref var firstLink = ref GetLinkDataPartReference(first);
            ref var secondLink = ref GetLinkDataPartReference(second);
            return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
             ↪  secondLink.Source, secondLink.Target);
        }

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

    /// <summary>
    /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
    ↪  (концом).
    /// </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>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public TLink Search(TLink source, TLink target)
    {
        var root = GetTreeRoot();
        while (!EqualToZero(root))
        {
            ref var rootLink = ref GetLinkDataPartReference(root);
            var rootSource = rootLink.Source;
            var rootTarget = rootLink.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 Zero;
    }

    // TODO: Return indices range instead of references count
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public TLink CountUsages(TLink link)
    {
        var root = GetTreeRoot();
        var total = GetSize(root);
        var totalRightIgnore = Zero;
        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 = Zero;
        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);
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
            ↪  EachUsageCore(@base, GetTreeRoot(), handler);

            // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
            ↪  low-level MSIL stack.
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
            {
                var @continue = Continue;
                if (EqualToZero(link))
                {
                    return @continue;
                }
                var linkBasePart = GetBasePartValue(link);
                var @break = Break;
                if (GreaterThan(linkBasePart, @base))
                {
                    if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
                    {
                        return @break;
                    }
                }
                else if (LessThan(linkBasePart, @base))
                {
                    if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
                    {
                        return @break;
                    }
                }
                else //if (linkBasePart == @base)
                {
                    if (AreEqual(handler(GetLinkValues(link)), @break))
                    {
                        return @break;
                    }
                    if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
                    {
                        return @break;
                    }
                    if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
                    {
                        return @break;
                    }
                }
                return @continue;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override void PrintNodeValue(TLink node, StringBuilder sb)
            {
                ref var link = ref GetLinkDataPartReference(node);
                sb.Append(' ');
                sb.Append(link.Source);
                sb.Append('-');
                sb.Append('>');
                sb.Append(link.Target);
            }
        }
    }
```

1.30 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSourcesSizeBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.Split.Generic
{
    public unsafe class ExternalLinksSourcesSizeBalancedTreeMethods<TLink> :
     ↪  ExternalLinksSizeBalancedTreeMethodsBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ExternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink> constants,
         ↪  byte* linksDataParts, byte* linksIndexParts, byte* header) : base(constants,
         ↪  linksDataParts, linksIndexParts, header) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLink GetLeftReference(TLink node) => ref
         ↪  GetLinkIndexPartReference(node).LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLink GetRightReference(TLink node) => ref
         ↪  GetLinkIndexPartReference(node).RightAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetLeft(TLink node) =>
         ↪  GetLinkIndexPartReference(node).LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetRight(TLink node) =>
         ↪  GetLinkIndexPartReference(node).RightAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(TLink node, TLink left) =>
         ↪  GetLinkIndexPartReference(node).LeftAsSource = left;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(TLink node, TLink right) =>
         ↪  GetLinkIndexPartReference(node).RightAsSource = right;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetSize(TLink node) =>
         ↪  GetLinkIndexPartReference(node).SizeAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLink node, TLink size) =>
         ↪  GetLinkIndexPartReference(node).SizeAsSource = size;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetTreeRoot() => GetHeaderReference().RootAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetBasePartValue(TLink link) =>
         ↪  GetLinkDataPartReference(link).Source;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
         ↪  TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
         ↪  AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
         ↪  TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
         ↪  AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLink node)
        {
            ref var link = ref GetLinkIndexPartReference(node);
            link.LeftAsSource = Zero;
            link.RightAsSource = Zero;
            link.SizeAsSource = Zero;
        }
    }
}
```

1.31 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksTargetsSizeBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.Split.Generic
{
    public unsafe class ExternalLinksTargetsSizeBalancedTreeMethods<TLink> :
     ↪  ExternalLinksSizeBalancedTreeMethodsBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ExternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink> constants,
        ↪  byte* linksDataParts, byte* linksIndexParts, byte* header) : base(constants,
        ↪  linksDataParts, linksIndexParts, header) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLink GetLeftReference(TLink node) => ref
        ↪  GetLinkIndexPartReference(node).LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLink GetRightReference(TLink node) => ref
        ↪  GetLinkIndexPartReference(node).RightAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetLeft(TLink node) =>
        ↪  GetLinkIndexPartReference(node).LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetRight(TLink node) =>
        ↪  GetLinkIndexPartReference(node).RightAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(TLink node, TLink left) =>
        ↪  GetLinkIndexPartReference(node).LeftAsTarget = left;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(TLink node, TLink right) =>
        ↪  GetLinkIndexPartReference(node).RightAsTarget = right;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetSize(TLink node) =>
        ↪  GetLinkIndexPartReference(node).SizeAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLink node, TLink size) =>
        ↪  GetLinkIndexPartReference(node).SizeAsTarget = size;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetTreeRoot() => GetHeaderReference().RootAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetBasePartValue(TLink link) =>
        ↪  GetLinkDataPartReference(link).Target;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
        ↪  TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
        ↪  AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
        ↪  TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
        ↪  AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLink node)
        {
            ref var link = ref GetLinkIndexPartReference(node);
            link.LeftAsTarget = Zero;
            link.RightAsTarget = Zero;
            link.SizeAsTarget = Zero;
        }
    }
}
```

1.32 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSizeBalancedTreeMethodsBase.cs

```
using System;
using System.Text;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections.Methods.Trees;
using Platform.Converters;
using static System.Runtime.CompilerServices.Unsafe;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.Split.Generic
{
    public unsafe abstract class InternalLinksSizeBalancedTreeMethodsBase<TLink> :
     ↪  SizeBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
    {
        private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
         ↪  UncheckedConverter<TLink, long>.Default;

        protected readonly TLink Break;
        protected readonly TLink Continue;
        protected readonly byte* LinksDataParts;
        protected readonly byte* LinksIndexParts;
        protected readonly byte* Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected InternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants,
         ↪  byte* linksDataParts, byte* linksIndexParts, byte* header)
        {
            LinksDataParts = linksDataParts;
            LinksIndexParts = linksIndexParts;
            Header = header;
            Break = constants.Break;
            Continue = constants.Continue;
        }

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
         ↪  AsRef<RawLinkDataPart<TLink>>(LinksDataParts + RawLinkDataPart<TLink>.SizeInBytes *
         ↪  _addressToInt64Converter.Convert(link));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
         ↪  ref AsRef<RawLinkIndexPart<TLink>>(LinksIndexParts +
         ↪  RawLinkIndexPart<TLink>.SizeInBytes * _addressToInt64Converter.Convert(link));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second) =>
         ↪  LessThan(GetKeyPartValue(first), GetKeyPartValue(second));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second) =>
         ↪  GreaterThan(GetKeyPartValue(first), GetKeyPartValue(second));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
        {
            ref var link = ref GetLinkDataPartReference(linkIndex);
            return new Link<TLink>(linkIndex, link.Source, link.Target);
        }

        public TLink this[TLink link, TLink index]
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get
            {
                var root = GetTreeRoot(link);
                if (GreaterOrEqualThan(index, GetSize(root)))
                {
                    return Zero;
                }
                while (!EqualToZero(root))
                {
                    var left = GetLeftOrDefault(root);
                    var leftSize = GetSizeOrZero(left);
                    if (LessThan(index, leftSize))
                    {
                        root = left;
                        continue;
                    }
                    if (AreEqual(index, leftSize))
                    {
                        return root;
```

```csharp
                    }
                    root = GetRightOrDefault(root);
                    index = Subtract(index, Increment(leftSize));
                }
                return Zero; // TODO: Impossible situation exception (only if tree structure
                 ↪  broken)
            }
        }

        /// <summary>
        /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
         ↪  (концом).
        /// </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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public abstract TLink Search(TLink source, TLink target);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected TLink SearchCore(TLink root, TLink key)
        {
            while (!EqualToZero(root))
            {
                var rootKey = GetKeyPartValue(root);
                if (LessThan(key, rootKey)) // node.Key < root.Key
                {
                    root = GetLeftOrDefault(root);
                }
                else if (GreaterThan(key, rootKey)) // node.Key > root.Key
                {
                    root = GetRightOrDefault(root);
                }
                else // node.Key == root.Key
                {
                    return root;
                }
            }
            return Zero;
        }

        // TODO: Return indices range instead of references count
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink CountUsages(TLink link) => GetSizeOrZero(GetTreeRoot(link));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
         ↪  EachUsageCore(@base, GetTreeRoot(@base), handler);

        // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
         ↪  low-level MSIL stack.
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
        {
            var @continue = Continue;
            if (EqualToZero(link))
            {
                return @continue;
            }
            var @break = Break;
            if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
            {
                return @break;
            }
            if (AreEqual(handler(GetLinkValues(link)), @break))
            {
                return @break;
            }
            if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
            {
                return @break;
            }
            return @continue;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void PrintNodeValue(TLink node, StringBuilder sb)
        {
            ref var link = ref GetLinkDataPartReference(node);
```

```
                sb.Append(' ');
                sb.Append(link.Source);
                sb.Append('-');
                sb.Append('>');
                sb.Append(link.Target);
            }
        }
    }
```

1.33 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSourcesSizeBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.Split.Generic
{
    public unsafe class InternalLinksSourcesSizeBalancedTreeMethods<TLink> :
     ↪  InternalLinksSizeBalancedTreeMethodsBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public InternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink> constants,
         ↪  byte* linksDataParts, byte* linksIndexParts, byte* header) : base(constants,
         ↪  linksDataParts, linksIndexParts, header) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLink GetLeftReference(TLink node) => ref
         ↪  GetLinkIndexPartReference(node).LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLink GetRightReference(TLink node) => ref
         ↪  GetLinkIndexPartReference(node).RightAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetLeft(TLink node) =>
         ↪  GetLinkIndexPartReference(node).LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetRight(TLink node) =>
         ↪  GetLinkIndexPartReference(node).RightAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(TLink node, TLink left) =>
         ↪  GetLinkIndexPartReference(node).LeftAsSource = left;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(TLink node, TLink right) =>
         ↪  GetLinkIndexPartReference(node).RightAsSource = right;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetSize(TLink node) =>
         ↪  GetLinkIndexPartReference(node).SizeAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLink node, TLink size) =>
         ↪  GetLinkIndexPartReference(node).SizeAsSource = size;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetTreeRoot(TLink link) =>
         ↪  GetLinkIndexPartReference(link).RootAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetBasePartValue(TLink link) =>
         ↪  GetLinkDataPartReference(link).Source;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetKeyPartValue(TLink link) =>
         ↪  GetLinkDataPartReference(link).Target;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLink node)
        {
            ref var link = ref GetLinkIndexPartReference(node);
            link.LeftAsSource = Zero;
            link.RightAsSource = Zero;
            link.SizeAsSource = Zero;
        }

        public override TLink Search(TLink source, TLink target) =>
         ↪  SearchCore(GetTreeRoot(source), target);
        }
}
```

1.34 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksTargetsSizeBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.Split.Generic
{
    public unsafe class InternalLinksTargetsSizeBalancedTreeMethods<TLink> :
     ↪  InternalLinksSizeBalancedTreeMethodsBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public InternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink> constants,
         ↪  byte* linksDataParts, byte* linksIndexParts, byte* header) : base(constants,
         ↪  linksDataParts, linksIndexParts, header) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLink GetLeftReference(TLink node) => ref
         ↪  GetLinkIndexPartReference(node).LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLink GetRightReference(TLink node) => ref
         ↪  GetLinkIndexPartReference(node).RightAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetLeft(TLink node) =>
         ↪  GetLinkIndexPartReference(node).LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetRight(TLink node) =>
         ↪  GetLinkIndexPartReference(node).RightAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(TLink node, TLink left) =>
         ↪  GetLinkIndexPartReference(node).LeftAsTarget = left;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(TLink node, TLink right) =>
         ↪  GetLinkIndexPartReference(node).RightAsTarget = right;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetSize(TLink node) =>
         ↪  GetLinkIndexPartReference(node).SizeAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLink node, TLink size) =>
         ↪  GetLinkIndexPartReference(node).SizeAsTarget = size;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetTreeRoot(TLink link) =>
         ↪  GetLinkIndexPartReference(link).RootAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetBasePartValue(TLink link) =>
         ↪  GetLinkDataPartReference(link).Target;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetKeyPartValue(TLink link) =>
         ↪  GetLinkDataPartReference(link).Source;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLink node)
        {
            ref var link = ref GetLinkIndexPartReference(node);
            link.LeftAsTarget = Zero;
            link.RightAsTarget = Zero;
            link.SizeAsTarget = Zero;
        }

        public override TLink Search(TLink source, TLink target) =>
         ↪  SearchCore(GetTreeRoot(target), source);
    }
}
```

1.35 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/SplitMemoryLinks.cs

```
using System;
using System.Runtime.CompilerServices;
using Platform.Singletons;
using Platform.Memory;
using static System.Runtime.CompilerServices.Unsafe;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.Split.Generic
{
    public unsafe class SplitMemoryLinks<TLink> : SplitMemoryLinksBase<TLink>
    {
        private readonly Func<ILinksTreeMethods<TLink>> _createInternalSourceTreeMethods;
        private readonly Func<ILinksTreeMethods<TLink>> _createExternalSourceTreeMethods;
        private readonly Func<ILinksTreeMethods<TLink>> _createInternalTargetTreeMethods;
        private readonly Func<ILinksTreeMethods<TLink>> _createExternalTargetTreeMethods;
        private byte* _header;
        private byte* _linksDataParts;
        private byte* _linksIndexParts;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
         ↪  indexMemory) : this(dataMemory, indexMemory, DefaultLinksSizeStep) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
         ↪  indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
         ↪  memoryReservationStep, Default<LinksConstants<TLink>>.Instance) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
         ↪  indexMemory, long memoryReservationStep, LinksConstants<TLink> constants) :
         ↪  base(dataMemory, indexMemory, memoryReservationStep, constants)
        {
            _createInternalSourceTreeMethods = () => new
             ↪  InternalLinksSourcesSizeBalancedTreeMethods<TLink>(Constants, _linksDataParts,
             ↪  _linksIndexParts, _header);
            _createExternalSourceTreeMethods = () => new
             ↪  ExternalLinksSourcesSizeBalancedTreeMethods<TLink>(Constants, _linksDataParts,
             ↪  _linksIndexParts, _header);
            _createInternalTargetTreeMethods = () => new
             ↪  InternalLinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _linksDataParts,
             ↪  _linksIndexParts, _header);
            _createExternalTargetTreeMethods = () => new
             ↪  ExternalLinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _linksDataParts,
             ↪  _linksIndexParts, _header);
            Init(dataMemory, indexMemory, memoryReservationStep);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetPointers(IResizableDirectMemory dataMemory,
         ↪  IResizableDirectMemory indexMemory)
        {
            _linksDataParts = (byte*)dataMemory.Pointer;
            _linksIndexParts = (byte*)indexMemory.Pointer;
            _header = _linksIndexParts;
            InternalSourcesTreeMethods = _createInternalSourceTreeMethods();
            ExternalSourcesTreeMethods = _createExternalSourceTreeMethods();
            InternalTargetsTreeMethods = _createInternalTargetTreeMethods();
            ExternalTargetsTreeMethods = _createExternalTargetTreeMethods();
            UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_linksDataParts, _header);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ResetPointers()
        {
            base.ResetPointers();
            _linksDataParts = null;
            _linksIndexParts = null;
            _header = null;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref LinksHeader<TLink> GetHeaderReference() => ref
         ↪  AsRef<LinksHeader<TLink>>(_header);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex)
         ↪  => ref AsRef<RawLinkDataPart<TLink>>(_linksDataParts + LinkDataPartSizeInBytes *
         ↪  ConvertToInt64(linkIndex));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
         ↪  linkIndex) => ref AsRef<RawLinkIndexPart<TLink>>(_linksIndexParts +
         ↪  LinkIndexPartSizeInBytes * ConvertToInt64(linkIndex));
    }
}
```

1.36 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/SplitMemoryLinksBase.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Disposables;
using Platform.Singletons;
using Platform.Converters;
using Platform.Numbers;
using Platform.Memory;
using Platform.Data.Exceptions;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.Split.Generic
{
    public abstract class SplitMemoryLinksBase<TLink> : DisposableBase, ILinks<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
         ↪  EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
        private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
         ↪  UncheckedConverter<TLink, long>.Default;
        private static readonly UncheckedConverter<long, TLink> _int64ToAddressConverter =
         ↪  UncheckedConverter<long, TLink>.Default;

        private static readonly TLink _zero = default;
        private static readonly TLink _one = Arithmetic.Increment(_zero);

        /// <summary>Возвращает размер одной связи в байтах.</summary>
        /// <remarks>
        /// Используется только во вне класса, не рекомедуется использовать внутри.
        /// Так как во вне не обязательно будет доступен unsafe C#.
        /// </remarks>
        public static readonly long LinkDataPartSizeInBytes = RawLinkDataPart<TLink>.SizeInBytes;

        public static readonly long LinkIndexPartSizeInBytes =
         ↪  RawLinkIndexPart<TLink>.SizeInBytes;

        public static readonly long LinkHeaderSizeInBytes = LinksHeader<TLink>.SizeInBytes;

        public static readonly long DefaultLinksSizeStep = 1 * 1024 * 1024;

        protected readonly IResizableDirectMemory _dataMemory;
        protected readonly IResizableDirectMemory _indexMemory;
        protected readonly long _dataMemoryReservationStepInBytes;
        protected readonly long _indexMemoryReservationStepInBytes;

        protected ILinksTreeMethods<TLink> InternalSourcesTreeMethods;
        protected ILinksTreeMethods<TLink> ExternalSourcesTreeMethods;
        protected ILinksTreeMethods<TLink> InternalTargetsTreeMethods;
        protected ILinksTreeMethods<TLink> ExternalTargetsTreeMethods;
        // 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
        protected ILinksListMethods<TLink> UnusedLinksListMethods;

        /// <summary>
        /// Возвращает общее число связей находящихся в хранилище.
        /// </summary>
        protected virtual TLink Total
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get
            {
                ref var header = ref GetHeaderReference();
                return Subtract(header.AllocatedLinks, header.FreeLinks);
            }
        }

        public virtual LinksConstants<TLink> Constants
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
        ↪  indexMemory, long memoryReservationStep, LinksConstants<TLink> constants)
        {
            _dataMemory = dataMemory;
            _indexMemory = indexMemory;
            _dataMemoryReservationStepInBytes = memoryReservationStep * LinkDataPartSizeInBytes;
            _indexMemoryReservationStepInBytes = memoryReservationStep *
            ↪  LinkIndexPartSizeInBytes;
            Constants = constants;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
        ↪  indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
        ↪  memoryReservationStep, Default<LinksConstants<TLink>>.Instance) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual void Init(IResizableDirectMemory dataMemory, IResizableDirectMemory
        ↪  indexMemory, long memoryReservationStep)
        {
            if (dataMemory.ReservedCapacity < memoryReservationStep)
            {
                dataMemory.ReservedCapacity = memoryReservationStep;
            }
            if (indexMemory.ReservedCapacity < memoryReservationStep)
            {
                indexMemory.ReservedCapacity = memoryReservationStep;
            }
            SetPointers(dataMemory, indexMemory);
            ref var header = ref GetHeaderReference();
            // Ensure correctness _memory.UsedCapacity over _header->AllocatedLinks
            // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
            dataMemory.UsedCapacity = ConvertToInt64(header.AllocatedLinks) *
            ↪  LinkDataPartSizeInBytes + LinkDataPartSizeInBytes; // First link is read only
            ↪  zero link.
            indexMemory.UsedCapacity = ConvertToInt64(header.AllocatedLinks) *
            ↪  LinkIndexPartSizeInBytes + LinkHeaderSizeInBytes;
            // Ensure correctness _memory.ReservedLinks over _header->ReservedCapacity
            // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
            header.ReservedLinks = ConvertToAddress((dataMemory.ReservedCapacity -
            ↪  LinkDataPartSizeInBytes) / LinkDataPartSizeInBytes);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Count(IList<TLink> restrictions)
        {
            // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
            if (restrictions.Count == 0)
            {
                return Total;
            }
            var constants = Constants;
            var any = constants.Any;
            var index = restrictions[constants.IndexPart];
            if (restrictions.Count == 1)
            {
                if (AreEqual(index, any))
                {
                    return Total;
                }
                return Exists(index) ? GetOne() : GetZero();
            }
            if (restrictions.Count == 2)
            {
                var value = restrictions[1];
                if (AreEqual(index, any))
                {
                    if (AreEqual(value, any))
                    {
                        return Total; // Any - как отсутствие ограничения
                    }
                    var externalReferencesRange = constants.ExternalReferencesRange;
                    if (externalReferencesRange.HasValue &&
                    ↪  externalReferencesRange.Value.Contains(value))
                    {
                        return Add(ExternalSourcesTreeMethods.CountUsages(value),
                        ↪  ExternalTargetsTreeMethods.CountUsages(value));
                    }
                    else
                    {
                        return Add(InternalSourcesTreeMethods.CountUsages(value), InternalTargetsTreeMethods.CountUsages(value));
                    }
                }
                else
                {
                    if (!Exists(index))
                    {
                        return GetZero();
                    }
                    if (AreEqual(value, any))
                    {
                        return GetOne();
                    }
                    ref var storedLinkValue = ref GetLinkDataPartReference(index);
                    if (AreEqual(storedLinkValue.Source, value) || AreEqual(storedLinkValue.Target, value))
                    {
                        return GetOne();
                    }
                    return GetZero();
                }
            }
            if (restrictions.Count == 3)
            {
                var externalReferencesRange = constants.ExternalReferencesRange;
                var source = restrictions[constants.SourcePart];
                var target = restrictions[constants.TargetPart];
                if (AreEqual(index, any))
                {
                    if (AreEqual(source, any) && AreEqual(target, any))
                    {
                        return Total;
                    }
                    else if (AreEqual(source, any))
                    {
                        if (externalReferencesRange.HasValue && externalReferencesRange.Value.Contains(target))
                        {
                            return ExternalTargetsTreeMethods.CountUsages(target);
                        }
                        else
                        {
                            return InternalTargetsTreeMethods.CountUsages(target);
                        }
                    }
                    else if (AreEqual(target, any))
                    {
                        if (externalReferencesRange.HasValue && externalReferencesRange.Value.Contains(source))
                        {
                            return ExternalSourcesTreeMethods.CountUsages(source);
                        }
                        else
                        {
                            return InternalSourcesTreeMethods.CountUsages(source);
                        }
                    }
                    else //if(source != Any && target != Any)
                    {
                        // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
                        TLink link;
                        if (externalReferencesRange.HasValue)
                        {
                            if (externalReferencesRange.Value.Contains(source) && externalReferencesRange.Value.Contains(target))
                            {
                                link = ExternalSourcesTreeMethods.Search(source, target);
                            }
                            else if (externalReferencesRange.Value.Contains(source))
                            {
                                link = InternalTargetsTreeMethods.Search(source, target);
                            }
                            else if (externalReferencesRange.Value.Contains(target))
                            {
                                link = InternalSourcesTreeMethods.Search(source, target);
                            }
                            else
                            {
                                if (GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
                                ↪   InternalTargetsTreeMethods.CountUsages(target)))
                                {
                                    link = InternalTargetsTreeMethods.Search(source, target);
                                }
                                else
                                {
                                    link = InternalSourcesTreeMethods.Search(source, target);
                                }
                            }
                        }
                        else
                        {
                            if (GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
                            ↪   InternalTargetsTreeMethods.CountUsages(target)))
                            {
                                link = InternalTargetsTreeMethods.Search(source, target);
                            }
                            else
                            {
                                link = InternalSourcesTreeMethods.Search(source, target);
                            }
                        }
                        return AreEqual(link, constants.Null) ? GetZero() : GetOne();
                    }
                }
                else
                {
                    if (!Exists(index))
                    {
                        return GetZero();
                    }
                    if (AreEqual(source, any) && AreEqual(target, any))
                    {
                        return GetOne();
                    }
                    ref var storedLinkValue = ref GetLinkDataPartReference(index);
                    if (!AreEqual(source, any) && !AreEqual(target, any))
                    {
                        if (AreEqual(storedLinkValue.Source, source) &&
                        ↪   AreEqual(storedLinkValue.Target, target))
                        {
                            return GetOne();
                        }
                        return GetZero();
                    }
                    var value = default(TLink);
                    if (AreEqual(source, any))
                    {
                        value = target;
                    }
                    if (AreEqual(target, any))
                    {
                        value = source;
                    }
                    if (AreEqual(storedLinkValue.Source, value) ||
                    ↪   AreEqual(storedLinkValue.Target, value))
                    {
                        return GetOne();
                    }
                    return GetZero();
                }
            }
            throw new NotSupportedException("Другие размеры и способы ограничений не
            ↪   поддерживаются.");
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
        {
            var constants = Constants;
            var @break = constants.Break;
            if (restrictions.Count == 0)
            {
                for (var link = GetOne(); LessOrEqualThan(link,
                 ↪  GetHeaderReference().AllocatedLinks); link = Increment(link))
                {
                    if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
                    {
                        return @break;
                    }
                }
                return @break;
            }
            var @continue = constants.Continue;
            var any = constants.Any;
            var index = restrictions[constants.IndexPart];
            if (restrictions.Count == 1)
            {
                if (AreEqual(index, any))
                {
                    return Each(handler, Array.Empty<TLink>());
                }
                if (!Exists(index))
                {
                    return @continue;
                }
                return handler(GetLinkStruct(index));
            }
            if (restrictions.Count == 2)
            {
                var value = restrictions[1];
                if (AreEqual(index, any))
                {
                    if (AreEqual(value, any))
                    {
                        return Each(handler, Array.Empty<TLink>());
                    }
                    if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
                    {
                        return @break;
                    }
                    return Each(handler, new Link<TLink>(index, any, value));
                }
                else
                {
                    if (!Exists(index))
                    {
                        return @continue;
                    }
                    if (AreEqual(value, any))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    ref var storedLinkValue = ref GetLinkDataPartReference(index);
                    if (AreEqual(storedLinkValue.Source, value) ||
                        AreEqual(storedLinkValue.Target, value))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    return @continue;
                }
            }
            if (restrictions.Count == 3)
            {
                var externalReferencesRange = constants.ExternalReferencesRange;
                var source = restrictions[constants.SourcePart];
                var target = restrictions[constants.TargetPart];
                if (AreEqual(index, any))
                {
                    if (AreEqual(source, any) && AreEqual(target, any))
                    {
                        return Each(handler, Array.Empty<TLink>());
                    }
                    else if (AreEqual(source, any))
                    {
                        if (externalReferencesRange.HasValue &&
                         ↪  externalReferencesRange.Value.Contains(target))
                        {
                            return ExternalTargetsTreeMethods.EachUsage(target, handler);
                        }
                        else
                    {
                        return InternalTargetsTreeMethods.EachUsage(target, handler);
                    }
                }
                else if (AreEqual(target, any))
                {
                    if (externalReferencesRange.HasValue &&
                    ↪   externalReferencesRange.Value.Contains(source))
                    {
                        return ExternalSourcesTreeMethods.EachUsage(source, handler);
                    }
                    else
                    {
                        return InternalSourcesTreeMethods.EachUsage(source, handler);
                    }
                }
                else //if(source != Any && target != Any)
                {
                    TLink link;
                    if (externalReferencesRange.HasValue)
                    {
                        if (externalReferencesRange.Value.Contains(source) &&
                        ↪   externalReferencesRange.Value.Contains(target))
                        {
                            link = ExternalSourcesTreeMethods.Search(source, target);
                        }
                        else if (externalReferencesRange.Value.Contains(source))
                        {
                            link = InternalTargetsTreeMethods.Search(source, target);
                        }
                        else if (externalReferencesRange.Value.Contains(target))
                        {
                            link = InternalSourcesTreeMethods.Search(source, target);
                        }
                        else
                        {
                            if (GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
                            ↪   InternalTargetsTreeMethods.CountUsages(target)))
                            {
                                link = InternalTargetsTreeMethods.Search(source, target);
                            }
                            else
                            {
                                link = InternalSourcesTreeMethods.Search(source, target);
                            }
                        }
                    }
                    else
                    {
                        if (GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
                        ↪   InternalTargetsTreeMethods.CountUsages(target)))
                        {
                            link = InternalTargetsTreeMethods.Search(source, target);
                        }
                        else
                        {
                            link = InternalSourcesTreeMethods.Search(source, target);
                        }
                    }
                    return AreEqual(link, constants.Null) ? @continue :
                    ↪   handler(GetLinkStruct(link));
                }
            }
            else
            {
                if (!Exists(index))
                {
                    return @continue;
                }
                if (AreEqual(source, any) && AreEqual(target, any))
                {
                    return handler(GetLinkStruct(index));
                }
                ref var storedLinkValue = ref GetLinkDataPartReference(index);
                if (!AreEqual(source, any) && !AreEqual(target, any))
                {
                    if (AreEqual(storedLinkValue.Source, source) &&
                        AreEqual(storedLinkValue.Target, target))
                        {
                            return handler(GetLinkStruct(index));
                        }
                        return @continue;
                    }
                    var value = default(TLink);
                    if (AreEqual(source, any))
                    {
                        value = target;
                    }
                    if (AreEqual(target, any))
                    {
                        value = source;
                    }
                    if (AreEqual(storedLinkValue.Source, value) ||
                        AreEqual(storedLinkValue.Target, value))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    return @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 virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
        {
            var constants = Constants;
            var @null = constants.Null;
            var externalReferencesRange = constants.ExternalReferencesRange;
            var linkIndex = restrictions[constants.IndexPart];
            ref var link = ref GetLinkDataPartReference(linkIndex);
            var source = link.Source;
            var target = link.Target;
            ref var header = ref GetHeaderReference();
            ref var rootAsSource = ref header.RootAsSource;
            ref var rootAsTarget = ref header.RootAsTarget;
            // Будет корректно работать только в том случае, если пространство выделенной связи
            ↪  предварительно заполнено нулями
            if (!AreEqual(source, @null))
            {
                if (externalReferencesRange.HasValue &&
                ↪  externalReferencesRange.Value.Contains(source))
                {
                    ExternalSourcesTreeMethods.Detach(ref rootAsSource, linkIndex);
                }
                else
                {
                    InternalSourcesTreeMethods.Detach(ref
                    ↪  GetLinkIndexPartReference(source).RootAsSource, linkIndex);
                }
            }
            if (!AreEqual(target, @null))
            {
                if (externalReferencesRange.HasValue &&
                ↪  externalReferencesRange.Value.Contains(target))
                {
                    ExternalTargetsTreeMethods.Detach(ref rootAsTarget, linkIndex);
                }
                else
                {
                    InternalTargetsTreeMethods.Detach(ref
                    ↪  GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
                }
            }
            source = link.Source = substitution[constants.SourcePart];
            target = link.Target = substitution[constants.TargetPart];
            if (!AreEqual(source, @null))
            {
                if (externalReferencesRange.HasValue &&
                ↪  externalReferencesRange.Value.Contains(source))
                {
```

```
                ExternalSourcesTreeMethods.Attach(ref rootAsSource, linkIndex);
            }
            else
            {
                InternalSourcesTreeMethods.Attach(ref
                 ↪  GetLinkIndexPartReference(source).RootAsSource, linkIndex);
            }
        }
        if (!AreEqual(target, @null))
        {
            if (externalReferencesRange.HasValue &&
             ↪  externalReferencesRange.Value.Contains(target))
            {
                ExternalTargetsTreeMethods.Attach(ref rootAsTarget, linkIndex);
            }
            else
            {
                InternalTargetsTreeMethods.Attach(ref
                 ↪  GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
            }
        }
        return linkIndex;
    }

    /// <remarks>
    /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
     ↪  пространство
    /// </remarks>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public virtual TLink Create(IList<TLink> restrictions)
    {
        ref var header = ref GetHeaderReference();
        var freeLink = header.FirstFreeLink;
        if (!AreEqual(freeLink, Constants.Null))
        {
            UnusedLinksListMethods.Detach(freeLink);
        }
        else
        {
            var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
            if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
            {
                throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
            }
            if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
            {
                _dataMemory.ReservedCapacity += _dataMemoryReservationStepInBytes;
                _indexMemory.ReservedCapacity += _indexMemoryReservationStepInBytes;
                SetPointers(_dataMemory, _indexMemory);
                header = ref GetHeaderReference();
                header.ReservedLinks = ConvertToAddress(_dataMemory.ReservedCapacity /
                 ↪  LinkDataPartSizeInBytes);
            }
            header.AllocatedLinks = Increment(header.AllocatedLinks);
            _dataMemory.UsedCapacity += LinkDataPartSizeInBytes;
            _indexMemory.UsedCapacity += LinkIndexPartSizeInBytes;
            freeLink = header.AllocatedLinks;
        }
        return freeLink;
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public virtual void Delete(IList<TLink> restrictions)
    {
        ref var header = ref GetHeaderReference();
        var link = restrictions[Constants.IndexPart];
        if (LessThan(link, header.AllocatedLinks))
        {
            UnusedLinksListMethods.AttachAsFirst(link);
        }
        else if (AreEqual(link, header.AllocatedLinks))
        {
            header.AllocatedLinks = Decrement(header.AllocatedLinks);
            _dataMemory.UsedCapacity -= LinkDataPartSizeInBytes;
            _indexMemory.UsedCapacity -= LinkIndexPartSizeInBytes;
            // 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)
```

```csharp
                while (GreaterThan(header.AllocatedLinks, GetZero()) && IsUnusedLink(header.AllocatedLinks))
                {
                    UnusedLinksListMethods.Detach(header.AllocatedLinks);
                    header.AllocatedLinks = Decrement(header.AllocatedLinks);
                    _dataMemory.UsedCapacity -= LinkDataPartSizeInBytes;
                    _indexMemory.UsedCapacity -= LinkIndexPartSizeInBytes;
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IList<TLink> GetLinkStruct(TLink linkIndex)
        {
            ref var link = ref GetLinkDataPartReference(linkIndex);
            return new Link<TLink>(linkIndex, link.Source, link.Target);
        }

        /// <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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetPointers(IResizableDirectMemory dataMemory, IResizableDirectMemory indexMemory);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual void ResetPointers()
        {
            InternalSourcesTreeMethods = null;
            ExternalSourcesTreeMethods = null;
            InternalTargetsTreeMethods = null;
            ExternalTargetsTreeMethods = null;
            UnusedLinksListMethods = null;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract ref LinksHeader<TLink> GetHeaderReference();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink linkIndex);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool Exists(TLink link)
            => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
            && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
            && !IsUnusedLink(link);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool IsUnusedLink(TLink linkIndex)
        {
            if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check is not needed
            {
                // TODO: Reduce access to memory in different location (should be enough to use just linkIndexPart)
                ref var linkDataPart = ref GetLinkDataPartReference(linkIndex);
                ref var linkIndexPart = ref GetLinkIndexPartReference(linkIndex);
                return AreEqual(linkIndexPart.SizeAsSource, default) && !AreEqual(linkDataPart.Source, default);
            }
            else
            {
                return true;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLink GetOne() => _one;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLink GetZero() => default;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool AreEqual(TLink first, TLink second) =>
         ↪  _equalityComparer.Equals(first, second);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool LessThan(TLink first, TLink second) => _comparer.Compare(first,
         ↪  second) < 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
         ↪  _comparer.Compare(first, second) <= 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool GreaterThan(TLink first, TLink second) =>
         ↪  _comparer.Compare(first, second) > 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
         ↪  _comparer.Compare(first, second) >= 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual long ConvertToInt64(TLink value) =>
         ↪  _addressToInt64Converter.Convert(value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLink ConvertToAddress(long value) =>
         ↪  _int64ToAddressConverter.Convert(value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
         ↪  second);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLink Subtract(TLink first, TLink second) =>
         ↪  Arithmetic<TLink>.Subtract(first, second);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);

        #region Disposable

        protected override bool AllowMultipleDisposeCalls
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => true;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                ResetPointers();
                _dataMemory.DisposeIfPossible();
                _indexMemory.DisposeIfPossible();
            }
        }

        #endregion
    }
}
```

1.37 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/UnusedLinksListMethods.cs

```
using System.Runtime.CompilerServices;
using Platform.Collections.Methods.Lists;
using Platform.Converters;
using static System.Runtime.CompilerServices.Unsafe;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.Split.Generic
{
    public unsafe class UnusedLinksListMethods<TLink> : CircularDoublyLinkedListMethods<TLink>,
     ↪  ILinksListMethods<TLink>
    {
        private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
        ↪  UncheckedConverter<TLink, long>.Default;

        private readonly byte* _links;
        private readonly byte* _header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnusedLinksListMethods(byte* links, byte* header)
        {
            _links = links;
            _header = header;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
        ↪  AsRef<LinksHeader<TLink>>(_header);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
        ↪  AsRef<RawLinkDataPart<TLink>>(_links + RawLinkDataPart<TLink>.SizeInBytes *
        ↪  _addressToInt64Converter.Convert(link));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetFirst() => GetHeaderReference().FirstFreeLink;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetLast() => GetHeaderReference().LastFreeLink;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetPrevious(TLink element) =>
        ↪  GetLinkDataPartReference(element).Source;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetNext(TLink element) =>
        ↪  GetLinkDataPartReference(element).Target;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetSize() => GetHeaderReference().FreeLinks;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetFirst(TLink element) => GetHeaderReference().FirstFreeLink =
        ↪  element;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLast(TLink element) => GetHeaderReference().LastFreeLink =
        ↪  element;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetPrevious(TLink element, TLink previous) =>
        ↪  GetLinkDataPartReference(element).Source = previous;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetNext(TLink element, TLink next) =>
        ↪  GetLinkDataPartReference(element).Target = next;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLink size) => GetHeaderReference().FreeLinks = size;
    }
}
```

1.38 ./csharp/Platform.Data.Doublets/Memory/Split/RawLinkDataPart.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.Split
{
    public struct RawLinkDataPart<TLink> : IEquatable<RawLinkDataPart<TLink>>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        public static readonly long SizeInBytes = Structure<RawLinkDataPart<TLink>>.Size;

        public TLink Source;
        public TLink Target;
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Equals(object obj) => obj is RawLinkDataPart<TLink> link ?
        ↪  Equals(link) : false;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Equals(RawLinkDataPart<TLink> other)
            => _equalityComparer.Equals(Source, other.Source)
            && _equalityComparer.Equals(Target, other.Target);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override int GetHashCode() => (Source, Target).GetHashCode();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator ==(RawLinkDataPart<TLink> left, RawLinkDataPart<TLink>
        ↪  right) => left.Equals(right);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator !=(RawLinkDataPart<TLink> left, RawLinkDataPart<TLink>
        ↪  right) => !(left == right);
    }
}
```

1.39 ./csharp/Platform.Data.Doublets/Memory/Split/RawLinkIndexPart.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.Split
{
    public struct RawLinkIndexPart<TLink> : IEquatable<RawLinkIndexPart<TLink>>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        public static readonly long SizeInBytes = Structure<RawLinkIndexPart<TLink>>.Size;

        public TLink RootAsSource;
        public TLink LeftAsSource;
        public TLink RightAsSource;
        public TLink SizeAsSource;
        public TLink RootAsTarget;
        public TLink LeftAsTarget;
        public TLink RightAsTarget;
        public TLink SizeAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Equals(object obj) => obj is RawLinkIndexPart<TLink> link ?
        ↪  Equals(link) : false;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Equals(RawLinkIndexPart<TLink> other)
            => _equalityComparer.Equals(RootAsSource, other.RootAsSource)
            && _equalityComparer.Equals(LeftAsSource, other.LeftAsSource)
            && _equalityComparer.Equals(RightAsSource, other.RightAsSource)
            && _equalityComparer.Equals(SizeAsSource, other.SizeAsSource)
            && _equalityComparer.Equals(RootAsTarget, other.RootAsTarget)
            && _equalityComparer.Equals(LeftAsTarget, other.LeftAsTarget)
            && _equalityComparer.Equals(RightAsTarget, other.RightAsTarget)
            && _equalityComparer.Equals(SizeAsTarget, other.SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override int GetHashCode() => (RootAsSource, LeftAsSource, RightAsSource,
        ↪  SizeAsSource, RootAsTarget, LeftAsTarget, RightAsTarget, SizeAsTarget).GetHashCode();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator ==(RawLinkIndexPart<TLink> left, RawLinkIndexPart<TLink>
        ↪  right) => left.Equals(right);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator !=(RawLinkIndexPart<TLink> left, RawLinkIndexPart<TLink>
        ↪  right) => !(left == right);
    }
}
```

1.40 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksAvlBalancedTreeMethodsBase.cs

```
using System;
using System.Text;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections.Methods.Trees;
using Platform.Converters;
using Platform.Numbers;
using static System.Runtime.CompilerServices.Unsafe;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Generic
{
    public unsafe abstract class LinksAvlBalancedTreeMethodsBase<TLink> :
     ↪  SizedAndThreadedAVLBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
    {
        private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
         ↪  UncheckedConverter<TLink, long>.Default;
        private static readonly UncheckedConverter<TLink, int> _addressToInt32Converter =
         ↪  UncheckedConverter<TLink, int>.Default;
        private static readonly UncheckedConverter<bool, TLink> _boolToAddressConverter =
         ↪  UncheckedConverter<bool, TLink>.Default;
        private static readonly UncheckedConverter<TLink, bool> _addressToBoolConverter =
         ↪  UncheckedConverter<TLink, bool>.Default;
        private static readonly UncheckedConverter<int, TLink> _int32ToAddressConverter =
         ↪  UncheckedConverter<int, TLink>.Default;

        protected readonly TLink Break;
        protected readonly TLink Continue;
        protected readonly byte* Links;
        protected readonly byte* Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected LinksAvlBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
         ↪  byte* header)
        {
            Links = links;
            Header = header;
            Break = constants.Break;
            Continue = constants.Continue;
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
         ↪  rootSource, TLink rootTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
         ↪  rootSource, TLink rootTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
         ↪  AsRef<LinksHeader<TLink>>(Header);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
         ↪  AsRef<RawLink<TLink>>(Links + RawLink<TLink>.SizeInBytes *
         ↪  _addressToInt64Converter.Convert(link));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
        {
            ref var link = ref GetLinkReference(linkIndex);
            return new Link<TLink>(linkIndex, link.Source, link.Target);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
        {
            ref var firstLink = ref GetLinkReference(first);
            ref var secondLink = ref GetLinkReference(second);
            return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
             ↪  secondLink.Source, secondLink.Target);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
        {
            ref var firstLink = ref GetLinkReference(first);
            ref var secondLink = ref GetLinkReference(second);
            return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
             ↪  secondLink.Source, secondLink.Target);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLink GetSizeValue(TLink value) => Bit<TLink>.PartialRead(value, 5,
        ↪  -5);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual void SetSizeValue(ref TLink storedValue, TLink size) => storedValue =
        ↪  Bit<TLink>.PartialWrite(storedValue, size, 5, -5);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool GetLeftIsChildValue(TLink value)
        {
            unchecked
            {
                return _addressToBoolConverter.Convert(Bit<TLink>.PartialRead(value, 4, 1));
                //return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 4, 1), default);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual void SetLeftIsChildValue(ref TLink storedValue, bool value)
        {
            unchecked
            {
                var previousValue = storedValue;
                var modified = Bit<TLink>.PartialWrite(previousValue,
                 ↪  _boolToAddressConverter.Convert(value), 4, 1);
                storedValue = modified;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool GetRightIsChildValue(TLink value)
        {
            unchecked
            {
                return _addressToBoolConverter.Convert(Bit<TLink>.PartialRead(value, 3, 1));
                //return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 3, 1), default);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual void SetRightIsChildValue(ref TLink storedValue, bool value)
        {
            unchecked
            {
                var previousValue = storedValue;
                var modified = Bit<TLink>.PartialWrite(previousValue,
                 ↪  _boolToAddressConverter.Convert(value), 3, 1);
                storedValue = modified;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected bool IsChild(TLink parent, TLink possibleChild)
        {
            var parentSize = GetSize(parent);
            var childSize = GetSizeOrZero(possibleChild);
            return GreaterThanZero(childSize) && LessOrEqualThan(childSize, parentSize);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual sbyte GetBalanceValue(TLink storedValue)
        {
            unchecked
            {
                var value = _addressToInt32Converter.Convert(Bit<TLink>.PartialRead(storedValue,
                 ↪  0, 3));
                value |= 0xF8 * ((value & 4) >> 2); // if negative, then continue ones to the
                ↪  end of sbyte
                return (sbyte)value;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual void SetBalanceValue(ref TLink storedValue, sbyte value)
        {
            unchecked
            {
                var packagedValue = _int32ToAddressConverter.Convert((byte)value >> 5 & 4 |
                ↪  value & 3);
                var modified = Bit<TLink>.PartialWrite(storedValue, packagedValue, 0, 3);
                storedValue = modified;
            }
        }

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

        /// <summary>
        /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
        ↪  (концом).
        /// </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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Search(TLink source, TLink target)
        {
            var root = GetTreeRoot();
            while (!EqualToZero(root))
            {
                ref var rootLink = ref GetLinkReference(root);
                var rootSource = rootLink.Source;
                var rootTarget = rootLink.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 Zero;
        }

        // TODO: Return indices range instead of references count
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink CountUsages(TLink link)
        {
            var root = GetTreeRoot();
            var total = GetSize(root);
            var totalRightIgnore = Zero;
            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 = Zero;
            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);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
        {
            var root = GetTreeRoot();
            if (EqualToZero(root))
            {
                return Continue;
            }
            TLink first = Zero, current = root;
            while (!EqualToZero(current))
            {
                var @base = GetBasePartValue(current);
                if (GreaterOrEqualThan(@base, link))
                {
                    if (AreEqual(@base, link))
                    {
                        first = current;
                    }
                    current = GetLeftOrDefault(current);
                }
                else
                {
                    current = GetRightOrDefault(current);
                }
            }
            if (!EqualToZero(first))
            {
                current = first;
                while (true)
                {
                    if (AreEqual(handler(GetLinkValues(current)), Break))
                    {
                        return Break;
                    }
                    current = GetNext(current);
                    if (EqualToZero(current) || !AreEqual(GetBasePartValue(current), link))
                    {
                        break;
                    }
                }
            }
            return Continue;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void PrintNodeValue(TLink node, StringBuilder sb)
        {
            ref var link = ref GetLinkReference(node);
            sb.Append(' ');
            sb.Append(link.Source);
            sb.Append('-');
            sb.Append('>');
            sb.Append(link.Target);
        }
    }
}
```

1.41 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSizeBalancedTreeMethodsBase.cs

```
using System;
using System.Text;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections.Methods.Trees;
using Platform.Converters;
using static System.Runtime.CompilerServices.Unsafe;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Generic
{
    public unsafe abstract class LinksSizeBalancedTreeMethodsBase<TLink> :
     ↪  SizeBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
    {
        private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
         ↪  UncheckedConverter<TLink, long>.Default;

        protected readonly TLink Break;
        protected readonly TLink Continue;
        protected readonly byte* Links;
        protected readonly byte* Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected LinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
         ↪  byte* header)
        {
            Links = links;
            Header = header;
            Break = constants.Break;
            Continue = constants.Continue;
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
         ↪  rootSource, TLink rootTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
         ↪  rootSource, TLink rootTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
         ↪  AsRef<LinksHeader<TLink>>(Header);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
         ↪  AsRef<RawLink<TLink>>(Links + RawLink<TLink>.SizeInBytes *
         ↪  _addressToInt64Converter.Convert(link));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
        {
            ref var link = ref GetLinkReference(linkIndex);
            return new Link<TLink>(linkIndex, link.Source, link.Target);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
        {
            ref var firstLink = ref GetLinkReference(first);
            ref var secondLink = ref GetLinkReference(second);
            return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
             ↪  secondLink.Source, secondLink.Target);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
        {
            ref var firstLink = ref GetLinkReference(first);
            ref var secondLink = ref GetLinkReference(second);
            return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
             ↪  secondLink.Source, secondLink.Target);
        }

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

        /// <summary>
        /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
         ↪  (концом).
        /// </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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Search(TLink source, TLink target)
        {
            var root = GetTreeRoot();
            while (!EqualToZero(root))
            {
                ref var rootLink = ref GetLinkReference(root);
                var rootSource = rootLink.Source;
                var rootTarget = rootLink.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 Zero;
        }

        // TODO: Return indices range instead of references count
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink CountUsages(TLink link)
        {
            var root = GetTreeRoot();
            var total = GetSize(root);
            var totalRightIgnore = Zero;
            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 = Zero;
            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);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
         ↪  EachUsageCore(@base, GetTreeRoot(), handler);

        // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
         ↪  low-level MSIL stack.
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
        {
            var @continue = Continue;
            if (EqualToZero(link))
            {
                return @continue;
            }
            var linkBasePart = GetBasePartValue(link);
            var @break = Break;
            if (GreaterThan(linkBasePart, @base))
            {
                if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
                {
                    return @break;
                }
            }
            else if (LessThan(linkBasePart, @base))
            {
                if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
                {
                    return @break;
                }
            }
```

```
                else //if (linkBasePart == @base)
                {
                    if (AreEqual(handler(GetLinkValues(link)), @break))
                    {
                        return @break;
                    }
                    if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
                    {
                        return @break;
                    }
                    if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
                    {
                        return @break;
                    }
                }
                return @continue;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override void PrintNodeValue(TLink node, StringBuilder sb)
            {
                ref var link = ref GetLinkReference(node);
                sb.Append(' ');
                sb.Append(link.Source);
                sb.Append('-');
                sb.Append('>');
                sb.Append(link.Target);
            }
        }
    }
```

1.42 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesAvlBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Generic
{
    public unsafe class LinksSourcesAvlBalancedTreeMethods<TLink> :
     ↪  LinksAvlBalancedTreeMethodsBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksSourcesAvlBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
         ↪  byte* header) : base(constants, links, header) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLink GetLeftReference(TLink node) => ref
         ↪  GetLinkReference(node).LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLink GetRightReference(TLink node) => ref
         ↪  GetLinkReference(node).RightAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(TLink node, TLink left) =>
         ↪  GetLinkReference(node).LeftAsSource = left;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(TLink node, TLink right) =>
         ↪  GetLinkReference(node).RightAsSource = right;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetSize(TLink node) =>
         ↪  GetSizeValue(GetLinkReference(node).SizeAsSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
         ↪  GetLinkReference(node).SizeAsSource, size);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetLeftIsChild(TLink node) =>
         ↪  GetLeftIsChildValue(GetLinkReference(node).SizeAsSource);

```

```csharp
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeftIsChild(TLink node, bool value) =>
         ↪  SetLeftIsChildValue(ref GetLinkReference(node).SizeAsSource, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetRightIsChild(TLink node) =>
         ↪  GetRightIsChildValue(GetLinkReference(node).SizeAsSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRightIsChild(TLink node, bool value) =>
         ↪  SetRightIsChildValue(ref GetLinkReference(node).SizeAsSource, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override sbyte GetBalance(TLink node) =>
         ↪  GetBalanceValue(GetLinkReference(node).SizeAsSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
         ↪  GetLinkReference(node).SizeAsSource, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetTreeRoot() => GetHeaderReference().RootAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
         ↪  TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
         ↪  AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
         ↪  TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
         ↪  AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLink node)
        {
            ref var link = ref GetLinkReference(node);
            link.LeftAsSource = Zero;
            link.RightAsSource = Zero;
            link.SizeAsSource = Zero;
        }
    }
}
```

1.43 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesSizeBalancedTreeMethods.cs

```csharp
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Generic
{
    public unsafe class LinksSourcesSizeBalancedTreeMethods<TLink> :
     ↪  LinksSizeBalancedTreeMethodsBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
         ↪  byte* header) : base(constants, links, header) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLink GetLeftReference(TLink node) => ref
         ↪  GetLinkReference(node).LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLink GetRightReference(TLink node) => ref
         ↪  GetLinkReference(node).RightAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(TLink node, TLink left) =>
         ↪  GetLinkReference(node).LeftAsSource = left;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(TLink node, TLink right) =>
        ↪  GetLinkReference(node).RightAsSource = right;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLink node, TLink size) =>
        ↪  GetLinkReference(node).SizeAsSource = size;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetTreeRoot() => GetHeaderReference().RootAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
        ↪  TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
        ↪  AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
        ↪  TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
        ↪  AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLink node)
        {
            ref var link = ref GetLinkReference(node);
            link.LeftAsSource = Zero;
            link.RightAsSource = Zero;
            link.SizeAsSource = Zero;
        }
    }
}
```

1.44 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsAvlBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Generic
{
    public unsafe class LinksTargetsAvlBalancedTreeMethods<TLink> :
    ↪  LinksAvlBalancedTreeMethodsBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksTargetsAvlBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
        ↪  byte* header) : base(constants, links, header) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLink GetLeftReference(TLink node) => ref
        ↪  GetLinkReference(node).LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLink GetRightReference(TLink node) => ref
        ↪  GetLinkReference(node).RightAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(TLink node, TLink left) =>
        ↪  GetLinkReference(node).LeftAsTarget = left;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(TLink node, TLink right) =>
        ↪  GetLinkReference(node).RightAsTarget = right;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetSize(TLink node) =>
        ↪  GetSizeValue(GetLinkReference(node).SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
         ↪  GetLinkReference(node).SizeAsTarget, size);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetLeftIsChild(TLink node) =>
         ↪  GetLeftIsChildValue(GetLinkReference(node).SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeftIsChild(TLink node, bool value) =>
         ↪  SetLeftIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetRightIsChild(TLink node) =>
         ↪  GetRightIsChildValue(GetLinkReference(node).SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRightIsChild(TLink node, bool value) =>
         ↪  SetRightIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override sbyte GetBalance(TLink node) =>
         ↪  GetBalanceValue(GetLinkReference(node).SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
         ↪  GetLinkReference(node).SizeAsTarget, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetTreeRoot() => GetHeaderReference().RootAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
         ↪  TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
         ↪  AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
         ↪  TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
         ↪  AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLink node)
        {
            ref var link = ref GetLinkReference(node);
            link.LeftAsTarget = Zero;
            link.RightAsTarget = Zero;
            link.SizeAsTarget = Zero;
        }
    }
}
```

1.45 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsSizeBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Generic
{
    public unsafe class LinksTargetsSizeBalancedTreeMethods<TLink> :
     ↪  LinksSizeBalancedTreeMethodsBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
         ↪  byte* header) : base(constants, links, header) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLink GetLeftReference(TLink node) => ref
         ↪  GetLinkReference(node).LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLink GetRightReference(TLink node) => ref
         ↪  GetLinkReference(node).RightAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsTarget;
```

```csharp

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(TLink node, TLink left) =>
         ↪  GetLinkReference(node).LeftAsTarget = left;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(TLink node, TLink right) =>
         ↪  GetLinkReference(node).RightAsTarget = right;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLink node, TLink size) =>
         ↪  GetLinkReference(node).SizeAsTarget = size;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetTreeRoot() => GetHeaderReference().RootAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
         ↪  TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
         ↪  AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
         ↪  TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
         ↪  AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLink node)
        {
            ref var link = ref GetLinkReference(node);
            link.LeftAsTarget = Zero;
            link.RightAsTarget = Zero;
            link.SizeAsTarget = Zero;
        }
    }
}
```

1.46 ./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinks.cs

```csharp
using System;
using System.Runtime.CompilerServices;
using Platform.Singletons;
using Platform.Memory;
using static System.Runtime.CompilerServices.Unsafe;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Generic
{
    public unsafe class UnitedMemoryLinks<TLink> : UnitedMemoryLinksBase<TLink>
    {
        private readonly Func<ILinksTreeMethods<TLink>> _createSourceTreeMethods;
        private readonly Func<ILinksTreeMethods<TLink>> _createTargetTreeMethods;
        private byte* _header;
        private byte* _links;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnitedMemoryLinks(string address) : this(address, DefaultLinksSizeStep) { }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
         ↪  DefaultLinksSizeStep) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnitedMemoryLinks(IResizableDirectMemory memory, long memoryReservationStep) :
         ↪  this(memory, memoryReservationStep, Default<LinksConstants<TLink>>.Instance, true) {
         ↪  }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnitedMemoryLinks(IResizableDirectMemory memory, long memoryReservationStep,
         ↪  LinksConstants<TLink> constants, bool useAvlBasedIndex) : base(memory,
         ↪  memoryReservationStep, constants)
        {
            if (useAvlBasedIndex)
            {
                _createSourceTreeMethods = () => new
                 ↪  LinksSourcesAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
                _createTargetTreeMethods = () => new
                 ↪  LinksTargetsAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
            }
            else
            {
                _createSourceTreeMethods = () => new
                 ↪  LinksSourcesSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
                _createTargetTreeMethods = () => new
                 ↪  LinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
            }
            Init(memory, memoryReservationStep);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetPointers(IResizableDirectMemory memory)
        {
            _links = (byte*)memory.Pointer;
            _header = _links;
            SourcesTreeMethods = _createSourceTreeMethods();
            TargetsTreeMethods = _createTargetTreeMethods();
            UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_links, _header);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ResetPointers()
        {
            base.ResetPointers();
            _links = null;
            _header = null;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref LinksHeader<TLink> GetHeaderReference() => ref
         ↪  AsRef<LinksHeader<TLink>>(_header);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLink<TLink> GetLinkReference(TLink linkIndex) => ref
         ↪  AsRef<RawLink<TLink>>(_links + LinkSizeInBytes * ConvertToInt64(linkIndex));
    }
}
```

1.47 ./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinksBase.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Disposables;
using Platform.Singletons;
using Platform.Converters;
using Platform.Numbers;
using Platform.Memory;
using Platform.Data.Exceptions;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Generic
{
    public abstract class UnitedMemoryLinksBase<TLink> : DisposableBase, ILinks<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
         ↪  EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
        private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
         ↪  UncheckedConverter<TLink, long>.Default;
        private static readonly UncheckedConverter<long, TLink> _int64ToAddressConverter =
         ↪  UncheckedConverter<long, TLink>.Default;

        private static readonly TLink _zero = default;
        private static readonly TLink _one = Arithmetic.Increment(_zero);

        /// <summary>Возвращает размер одной связи в байтах.</summary>
        /// <remarks>
        /// Используется только во вне класса, не рекомедуется использовать внутри.
        /// Так как во вне не обязательно будет доступен unsafe C#.
        /// </remarks>
        public static readonly long LinkSizeInBytes = RawLink<TLink>.SizeInBytes;

        public static readonly long LinkHeaderSizeInBytes = LinksHeader<TLink>.SizeInBytes;

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

        protected readonly IResizableDirectMemory _memory;
        protected readonly long _memoryReservationStep;

        protected ILinksTreeMethods<TLink> TargetsTreeMethods;
        protected ILinksTreeMethods<TLink> SourcesTreeMethods;
        // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
         ↪  нужно использовать не список а дерево, так как так можно быстрее проверить на
         ↪  наличие связи внутри
        protected ILinksListMethods<TLink> UnusedLinksListMethods;

        /// <summary>
        /// Возвращает общее число связей находящихся в хранилище.
        /// </summary>
        protected virtual TLink Total
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get
            {
                ref var header = ref GetHeaderReference();
                return Subtract(header.AllocatedLinks, header.FreeLinks);
            }
        }

        public virtual LinksConstants<TLink> Constants
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected UnitedMemoryLinksBase(IResizableDirectMemory memory, long
         ↪  memoryReservationStep, LinksConstants<TLink> constants)
        {
            _memory = memory;
            _memoryReservationStep = memoryReservationStep;
            Constants = constants;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected UnitedMemoryLinksBase(IResizableDirectMemory memory, long
         ↪  memoryReservationStep) : this(memory, memoryReservationStep,
         ↪  Default<LinksConstants<TLink>>.Instance) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual void Init(IResizableDirectMemory memory, long memoryReservationStep)
        {
            if (memory.ReservedCapacity < memoryReservationStep)
            {
                memory.ReservedCapacity = memoryReservationStep;
            }
            SetPointers(memory);
            ref var header = ref GetHeaderReference();
            // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
            memory.UsedCapacity = ConvertToInt64(header.AllocatedLinks) * LinkSizeInBytes +
             ↪  LinkHeaderSizeInBytes;
            // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
            header.ReservedLinks = ConvertToAddress((memory.ReservedCapacity -
             ↪  LinkHeaderSizeInBytes) / LinkSizeInBytes);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Count(IList<TLink> restrictions)
        {
            // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
            if (restrictions.Count == 0)
            {
                return Total;
            }
            var constants = Constants;
            var any = constants.Any;
            var index = restrictions[constants.IndexPart];
            if (restrictions.Count == 1)
            {
                if (AreEqual(index, any))
                {
                    return Total;
                }
                return Exists(index) ? GetOne() : GetZero();
            }
            if (restrictions.Count == 2)
            {
                var value = restrictions[1];
                if (AreEqual(index, any))
                {
                    if (AreEqual(value, any))
                    {
                        return Total; // Any - как отсутствие ограничения
                    }
                    return Add(SourcesTreeMethods.CountUsages(value),
                    ↪  TargetsTreeMethods.CountUsages(value));
                }
                else
                {
                    if (!Exists(index))
                    {
                        return GetZero();
                    }
                    if (AreEqual(value, any))
                    {
                        return GetOne();
                    }
                    ref var storedLinkValue = ref GetLinkReference(index);
                    if (AreEqual(storedLinkValue.Source, value) ||
                    ↪  AreEqual(storedLinkValue.Target, value))
                    {
                        return GetOne();
                    }
                    return GetZero();
                }
            }
            if (restrictions.Count == 3)
            {
                var source = restrictions[constants.SourcePart];
                var target = restrictions[constants.TargetPart];
                if (AreEqual(index, any))
                {
                    if (AreEqual(source, any) && AreEqual(target, any))
                    {
                        return Total;
                    }
                    else if (AreEqual(source, any))
                    {
                        return TargetsTreeMethods.CountUsages(target);
                    }
                    else if (AreEqual(target, any))
                    {
                        return SourcesTreeMethods.CountUsages(source);
                    }
                    else //if(source != Any && target != Any)
                    {
                        // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
                        var link = SourcesTreeMethods.Search(source, target);
                        return AreEqual(link, constants.Null) ? GetZero() : GetOne();
                    }
                }
                else
                {
                    if (!Exists(index))
                    {
                        return GetZero();
                    }
                    if (AreEqual(source, any) && AreEqual(target, any))
                    {
                        return GetOne();
                    }
                    ref var storedLinkValue = ref GetLinkReference(index);
                    if (!AreEqual(source, any) && !AreEqual(target, any))
                    {
                        if (AreEqual(storedLinkValue.Source, source) &&
                         ↪  AreEqual(storedLinkValue.Target, target))
                        {
                            return GetOne();
                        }
                        return GetZero();
                    }
                    var value = default(TLink);
                    if (AreEqual(source, any))
                    {
                        value = target;
                    }
                    if (AreEqual(target, any))
                    {
                        value = source;
                    }
                    if (AreEqual(storedLinkValue.Source, value) ||
                     ↪  AreEqual(storedLinkValue.Target, value))
                    {
                        return GetOne();
                    }
                    return GetZero();
                }
            }
            throw new NotSupportedException("Другие размеры и способы ограничений не
             ↪  поддерживаются.");
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
        {
            var constants = Constants;
            var @break = constants.Break;
            if (restrictions.Count == 0)
            {
                for (var link = GetOne(); LessOrEqualThan(link,
                 ↪  GetHeaderReference().AllocatedLinks); link = Increment(link))
                {
                    if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
                    {
                        return @break;
                    }
                }
                return @break;
            }
            var @continue = constants.Continue;
            var any = constants.Any;
            var index = restrictions[constants.IndexPart];
            if (restrictions.Count == 1)
            {
                if (AreEqual(index, any))
                {
                    return Each(handler, Array.Empty<TLink>());
                }
                if (!Exists(index))
                {
                    return @continue;
                }
                return handler(GetLinkStruct(index));
            }
            if (restrictions.Count == 2)
            {
                var value = restrictions[1];
                if (AreEqual(index, any))
                {
                    if (AreEqual(value, any))
                    {
                        return Each(handler, Array.Empty<TLink>());
                    }
                    if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
                    {
                        return @break;
                    }
                    return Each(handler, new Link<TLink>(index, any, value));
                }
                else
                {
                    if (!Exists(index))
                    {
                        return @continue;
                    }
                    if (AreEqual(value, any))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    ref var storedLinkValue = ref GetLinkReference(index);
                    if (AreEqual(storedLinkValue.Source, value) ||
                        AreEqual(storedLinkValue.Target, value))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    return @continue;
                }
            }
            if (restrictions.Count == 3)
            {
                var source = restrictions[constants.SourcePart];
                var target = restrictions[constants.TargetPart];
                if (AreEqual(index, any))
                {
                    if (AreEqual(source, any) && AreEqual(target, any))
                    {
                        return Each(handler, Array.Empty<TLink>());
                    }
                    else if (AreEqual(source, any))
                    {
                        return TargetsTreeMethods.EachUsage(target, handler);
                    }
                    else if (AreEqual(target, any))
                    {
                        return SourcesTreeMethods.EachUsage(source, handler);
                    }
                    else //if(source != Any && target != Any)
                    {
                        var link = SourcesTreeMethods.Search(source, target);
                        return AreEqual(link, constants.Null) ? @continue :
                         ↪  handler(GetLinkStruct(link));
                    }
                }
                else
                {
                    if (!Exists(index))
                    {
                        return @continue;
                    }
                    if (AreEqual(source, any) && AreEqual(target, any))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    ref var storedLinkValue = ref GetLinkReference(index);
                    if (!AreEqual(source, any) && !AreEqual(target, any))
                    {
                        if (AreEqual(storedLinkValue.Source, source) &&
                            AreEqual(storedLinkValue.Target, target))
                        {
                            return handler(GetLinkStruct(index));
                        }
                        return @continue;
                    }
                    var value = default(TLink);
                    if (AreEqual(source, any))
                    {
                        value = target;
                    }
                    if (AreEqual(target, any))
                    {
                        value = source;
                    }
                    if (AreEqual(storedLinkValue.Source, value) ||
                        AreEqual(storedLinkValue.Target, value))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    return @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 virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
        {
            var constants = Constants;
            var @null = constants.Null;
            var linkIndex = restrictions[constants.IndexPart];
            ref var link = ref GetLinkReference(linkIndex);
            ref var header = ref GetHeaderReference();
            ref var firstAsSource = ref header.RootAsSource;
            ref var firstAsTarget = ref header.RootAsTarget;
            // Будет корректно работать только в том случае, если пространство выделенной связи
             ↪  предварительно заполнено нулями
            if (!AreEqual(link.Source, @null))
            {
                SourcesTreeMethods.Detach(ref firstAsSource, linkIndex);
            }
            if (!AreEqual(link.Target, @null))
            {
                TargetsTreeMethods.Detach(ref firstAsTarget, linkIndex);
            }
            link.Source = substitution[constants.SourcePart];
            link.Target = substitution[constants.TargetPart];
            if (!AreEqual(link.Source, @null))
            {
                SourcesTreeMethods.Attach(ref firstAsSource, linkIndex);
            }
            if (!AreEqual(link.Target, @null))
            {
                TargetsTreeMethods.Attach(ref firstAsTarget, linkIndex);
            }
            return linkIndex;
        }

        /// <remarks>
        /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
         ↪  пространство
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Create(IList<TLink> restrictions)
        {
            ref var header = ref GetHeaderReference();
            var freeLink = header.FirstFreeLink;
            if (!AreEqual(freeLink, Constants.Null))
            {
                UnusedLinksListMethods.Detach(freeLink);
            }
            else
            {
                var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
                if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
                {
                    throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
                }
                if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
                {
                    _memory.ReservedCapacity += _memoryReservationStep;
                    SetPointers(_memory);
                    header = ref GetHeaderReference();
                    header.ReservedLinks = ConvertToAddress(_memory.ReservedCapacity /
                     ↪  LinkSizeInBytes);
                }
                header.AllocatedLinks = Increment(header.AllocatedLinks);
                _memory.UsedCapacity += LinkSizeInBytes;
                freeLink = header.AllocatedLinks;
            }
            return freeLink;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual void Delete(IList<TLink> restrictions)
        {
            ref var header = ref GetHeaderReference();
            var link = restrictions[Constants.IndexPart];
            if (LessThan(link, header.AllocatedLinks))
            {
                UnusedLinksListMethods.AttachAsFirst(link);
            }
            else if (AreEqual(link, header.AllocatedLinks))
            {
                header.AllocatedLinks = Decrement(header.AllocatedLinks);
                _memory.UsedCapacity -= LinkSizeInBytes;
                // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
                ↪  пока не дойдём до первой существующей связи
                // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
                while (GreaterThan(header.AllocatedLinks, GetZero()) &&
                ↪  IsUnusedLink(header.AllocatedLinks))
                {
                    UnusedLinksListMethods.Detach(header.AllocatedLinks);
                    header.AllocatedLinks = Decrement(header.AllocatedLinks);
                    _memory.UsedCapacity -= LinkSizeInBytes;
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IList<TLink> GetLinkStruct(TLink linkIndex)
        {
            ref var link = ref GetLinkReference(linkIndex);
            return new Link<TLink>(linkIndex, link.Source, link.Target);
        }

        /// <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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetPointers(IResizableDirectMemory memory);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual void ResetPointers()
        {
            SourcesTreeMethods = null;
            TargetsTreeMethods = null;
            UnusedLinksListMethods = null;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract ref LinksHeader<TLink> GetHeaderReference();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract ref RawLink<TLink> GetLinkReference(TLink linkIndex);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool Exists(TLink link)
            => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
            && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
            && !IsUnusedLink(link);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool IsUnusedLink(TLink linkIndex)
        {
            if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
            ↪  is not needed
            {
                ref var link = ref GetLinkReference(linkIndex);
                return AreEqual(link.SizeAsSource, default) && !AreEqual(link.Source, default);
            }
            else
            {
                return true;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLink GetOne() => _one;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLink GetZero() => default;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool AreEqual(TLink first, TLink second) =>
         ↪  _equalityComparer.Equals(first, second);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool LessThan(TLink first, TLink second) => _comparer.Compare(first,
         ↪  second) < 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
         ↪  _comparer.Compare(first, second) <= 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool GreaterThan(TLink first, TLink second) =>
         ↪  _comparer.Compare(first, second) > 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
         ↪  _comparer.Compare(first, second) >= 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual long ConvertToInt64(TLink value) =>
         ↪  _addressToInt64Converter.Convert(value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLink ConvertToAddress(long value) =>
         ↪  _int64ToAddressConverter.Convert(value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
         ↪  second);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLink Subtract(TLink first, TLink second) =>
         ↪  Arithmetic<TLink>.Subtract(first, second);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);

        #region Disposable

        protected override bool AllowMultipleDisposeCalls
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => true;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                ResetPointers();
                _memory.DisposeIfPossible();
            }
        }

        #endregion
    }
}
```

1.48 ./csharp/Platform.Data.Doublets/Memory/United/Generic/UnusedLinksListMethods.cs

```
using System.Runtime.CompilerServices;
using Platform.Collections.Methods.Lists;
using Platform.Converters;
using static System.Runtime.CompilerServices.Unsafe;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Generic
{
    public unsafe class UnusedLinksListMethods<TLink> : CircularDoublyLinkedListMethods<TLink>,
     ↪  ILinksListMethods<TLink>
    {
        private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
         ↪  UncheckedConverter<TLink, long>.Default;

        private readonly byte* _links;
        private readonly byte* _header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnusedLinksListMethods(byte* links, byte* header)
        {
            _links = links;
            _header = header;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
         ↪  AsRef<LinksHeader<TLink>>(_header);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
         ↪  AsRef<RawLink<TLink>>(_links + RawLink<TLink>.SizeInBytes *
         ↪  _addressToInt64Converter.Convert(link));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetFirst() => GetHeaderReference().FirstFreeLink;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetLast() => GetHeaderReference().LastFreeLink;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetPrevious(TLink element) => GetLinkReference(element).Source;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetNext(TLink element) => GetLinkReference(element).Target;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetSize() => GetHeaderReference().FreeLinks;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetFirst(TLink element) => GetHeaderReference().FirstFreeLink =
         ↪  element;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLast(TLink element) => GetHeaderReference().LastFreeLink =
         ↪  element;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetPrevious(TLink element, TLink previous) =>
         ↪  GetLinkReference(element).Source = previous;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetNext(TLink element, TLink next) =>
         ↪  GetLinkReference(element).Target = next;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLink size) => GetHeaderReference().FreeLinks = size;
    }
}
```

1.49 ./csharp/Platform.Data.Doublets/Memory/United/RawLink.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United
```

```
{
    public struct RawLink<TLink> : IEquatable<RawLink<TLink>>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        public static readonly long SizeInBytes = Structure<RawLink<TLink>>.Size;

        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 override bool Equals(object obj) => obj is RawLink<TLink> link ? Equals(link) :
        ↪  false;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Equals(RawLink<TLink> other)
            => _equalityComparer.Equals(Source, other.Source)
            && _equalityComparer.Equals(Target, other.Target)
            && _equalityComparer.Equals(LeftAsSource, other.LeftAsSource)
            && _equalityComparer.Equals(RightAsSource, other.RightAsSource)
            && _equalityComparer.Equals(SizeAsSource, other.SizeAsSource)
            && _equalityComparer.Equals(LeftAsTarget, other.LeftAsTarget)
            && _equalityComparer.Equals(RightAsTarget, other.RightAsTarget)
            && _equalityComparer.Equals(SizeAsTarget, other.SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override int GetHashCode() => (Source, Target, LeftAsSource, RightAsSource,
        ↪  SizeAsSource, LeftAsTarget, RightAsTarget, SizeAsTarget).GetHashCode();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator ==(RawLink<TLink> left, RawLink<TLink> right) =>
        ↪  left.Equals(right);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator !=(RawLink<TLink> left, RawLink<TLink> right) => !(left ==
        ↪  right);
    }
}
```

1.50 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksAvlBalancedTreeMethodsBase.cs

```
using System.Runtime.CompilerServices;
using Platform.Data.Doublets.Memory.United.Generic;
using static System.Runtime.CompilerServices.Unsafe;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Specific
{
    public unsafe abstract class UInt64LinksAvlBalancedTreeMethodsBase :
    ↪  LinksAvlBalancedTreeMethodsBase<ulong>
    {
        protected new readonly RawLink<ulong>* Links;
        protected new readonly LinksHeader<ulong>* Header;

        protected UInt64LinksAvlBalancedTreeMethodsBase(LinksConstants<ulong> constants,
        ↪  RawLink<ulong>* links, LinksHeader<ulong>* header)
            : base(constants, (byte*)links, (byte*)header)
        {
            Links = links;
            Header = header;
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool AreEqual(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 == 0UL; // 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;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
        {
            ref var firstLink = ref Links[first];
            ref var secondLink = ref Links[second];
            return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
             ↪  secondLink.Source, secondLink.Target);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
        {
            ref var firstLink = ref Links[first];
            ref var secondLink = ref Links[second];
            return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
             ↪  secondLink.Source, secondLink.Target);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetSizeValue(ulong value) => (value & 4294967264UL) >> 5;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSizeValue(ref ulong storedValue, ulong size) => storedValue =
        ↪  storedValue & 31UL | (size & 134217727UL) << 5;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetLeftIsChildValue(ulong value) => (value & 16UL) >> 4 == 1UL;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeftIsChildValue(ref ulong storedValue, bool value) =>
        ↪  storedValue = storedValue & 4294967279UL | (As<bool, byte>(ref value) & 1UL) << 4;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetRightIsChildValue(ulong value) => (value & 8UL) >> 3 == 1UL;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRightIsChildValue(ref ulong storedValue, bool value) =>
        ↪  storedValue = storedValue & 4294967287UL | (As<bool, byte>(ref value) & 1UL) << 3;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override sbyte GetBalanceValue(ulong value) => unchecked((sbyte)(value & 7UL |
        ↪  0xF8UL * ((value & 4UL) >> 2))); // if negative, then continue ones to the end of
        ↪  sbyte

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetBalanceValue(ref ulong storedValue, sbyte value) =>
        ↪  storedValue = unchecked(storedValue & 4294967288UL | (ulong)((byte)value >> 5 & 4 |
        ↪  value & 3) & 7UL);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
    }
}
```

1.51 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSizeBalancedTreeMethodsBase.cs

```
using System.Runtime.CompilerServices;
using Platform.Data.Doublets.Memory.United.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Specific
{
    public unsafe abstract class UInt64LinksSizeBalancedTreeMethodsBase :
    ↪  LinksSizeBalancedTreeMethodsBase<ulong>
    {
        protected new readonly RawLink<ulong>* Links;
        protected new readonly LinksHeader<ulong>* Header;

        protected UInt64LinksSizeBalancedTreeMethodsBase(LinksConstants<ulong> constants,
        ↪  RawLink<ulong>* links, LinksHeader<ulong>* header)
            : base(constants, (byte*)links, (byte*)header)
        {
            Links = links;
            Header = header;
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool AreEqual(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 == 0UL; // 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;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
        {
            ref var firstLink = ref Links[first];
            ref var secondLink = ref Links[second];
            return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
             ↪  secondLink.Source, secondLink.Target);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
        {
            ref var firstLink = ref Links[first];
            ref var secondLink = ref Links[second];
            return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
             ↪  secondLink.Source, secondLink.Target);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
    }
}
```

1.52 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesAvlBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Specific
{
    public unsafe class UInt64LinksSourcesAvlBalancedTreeMethods :
     ↪  UInt64LinksAvlBalancedTreeMethodsBase
    {
        public UInt64LinksSourcesAvlBalancedTreeMethods(LinksConstants<ulong> constants,
         ↪  RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
         ↪  { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetLeftReference(ulong node) => ref
         ↪  Links[node].LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetRightReference(ulong node) => ref
         ↪  Links[node].RightAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetRight(ulong node) => Links[node].RightAsSource;

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetSize(ulong node) => GetSizeValue(Links[node].SizeAsSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
         ↪  Links[node].SizeAsSource, size);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetLeftIsChild(ulong node) =>
         ↪  GetLeftIsChildValue(Links[node].SizeAsSource);

        //[MethodImpl(MethodImplOptions.AggressiveInlining)]
        //protected override bool GetLeftIsChild(ulong node) => IsChild(node, GetLeft(node));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeftIsChild(ulong node, bool value) =>
         ↪  SetLeftIsChildValue(ref Links[node].SizeAsSource, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetRightIsChild(ulong node) =>
         ↪  GetRightIsChildValue(Links[node].SizeAsSource);

        //[MethodImpl(MethodImplOptions.AggressiveInlining)]
        //protected override bool GetRightIsChild(ulong node) => IsChild(node, GetRight(node));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRightIsChild(ulong node, bool value) =>
         ↪  SetRightIsChildValue(ref Links[node].SizeAsSource, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override sbyte GetBalance(ulong node) =>
         ↪  GetBalanceValue(Links[node].SizeAsSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
         ↪  Links[node].SizeAsSource, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetTreeRoot() => Header->RootAsSource;

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override 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)
        {
            ref var link = ref Links[node];
            link.LeftAsSource = 0UL;
            link.RightAsSource = 0UL;
            link.SizeAsSource = 0UL;
        }
    }
}
```

1.53 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesSizeBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Specific
{
    public unsafe class UInt64LinksSourcesSizeBalancedTreeMethods :
     ↪  UInt64LinksSizeBalancedTreeMethodsBase
    {
        public UInt64LinksSourcesSizeBalancedTreeMethods(LinksConstants<ulong> constants,
         ↪  RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
         ↪  { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetLeftReference(ulong node) => ref
         ↪  Links[node].LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetRightReference(ulong node) => ref
         ↪  Links[node].RightAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetRight(ulong node) => Links[node].RightAsSource;

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetSize(ulong node) => Links[node].SizeAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsSource =
        ↪  size;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetTreeRoot() => Header->RootAsSource;

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override 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)
        {
            ref var link = ref Links[node];
            link.LeftAsSource = 0UL;
            link.RightAsSource = 0UL;
            link.SizeAsSource = 0UL;
        }
    }
}
```

1.54 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsAvlBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Specific
{
    public unsafe class UInt64LinksTargetsAvlBalancedTreeMethods :
    ↪  UInt64LinksAvlBalancedTreeMethodsBase
    {
        public UInt64LinksTargetsAvlBalancedTreeMethods(LinksConstants<ulong> constants,
        ↪  RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
        ↪  { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetLeftReference(ulong node) => ref
        ↪  Links[node].LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetRightReference(ulong node) => ref
        ↪  Links[node].RightAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetSize(ulong node) => GetSizeValue(Links[node].SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
         ↪  Links[node].SizeAsTarget, size);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetLeftIsChild(ulong node) =>
         ↪  GetLeftIsChildValue(Links[node].SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeftIsChild(ulong node, bool value) =>
         ↪  SetLeftIsChildValue(ref Links[node].SizeAsTarget, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetRightIsChild(ulong node) =>
         ↪  GetRightIsChildValue(Links[node].SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRightIsChild(ulong node, bool value) =>
         ↪  SetRightIsChildValue(ref Links[node].SizeAsTarget, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override sbyte GetBalance(ulong node) =>
         ↪  GetBalanceValue(Links[node].SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
         ↪  Links[node].SizeAsTarget, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetTreeRoot() => Header->RootAsTarget;

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

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

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

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

1.55 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsSizeBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Specific
{
    public unsafe class UInt64LinksTargetsSizeBalancedTreeMethods :
     ↪  UInt64LinksSizeBalancedTreeMethodsBase
    {
        public UInt64LinksTargetsSizeBalancedTreeMethods(LinksConstants<ulong> constants,
        ↪  RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
        ↪  { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetLeftReference(ulong node) => ref
        ↪  Links[node].LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetRightReference(ulong node) => ref
        ↪  Links[node].RightAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetTreeRoot() => Header->RootAsTarget;

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

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

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

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

1.56 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnitedMemoryLinks.cs

```
using System;
using System.Runtime.CompilerServices;
using Platform.Memory;
using Platform.Singletons;
using Platform.Data.Doublets.Memory.United.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Specific
{
    /// <summary>
    /// <para>Represents a low-level implementation of direct access to resizable memory, for
     ↪  organizing the storage of links with addresses represented as <see cref="ulong"
     ↪  />.</para>
    /// <para>Представляет низкоуровневую реализация прямого доступа к памяти с переменным
    ↪  размером, для организации хранения связей с адресами представленными в виде <see
    ↪  cref="ulong"/>.</para>
    /// </summary>
    public unsafe class UInt64UnitedMemoryLinks : UnitedMemoryLinksBase<ulong>
    {
        private readonly Func<ILinksTreeMethods<ulong>> _createSourceTreeMethods;
        private readonly Func<ILinksTreeMethods<ulong>> _createTargetTreeMethods;
        private LinksHeader<ulong>* _header;
        private RawLink<ulong>* _links;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64UnitedMemoryLinks(string address) : this(address, DefaultLinksSizeStep) { }

        /// <summary>
        /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
        ↪  минимальным шагом расширения базы данных.
        /// </summary>
        /// <param name="address">Полный пусть к файлу базы данных.</param>
        /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
        ↪  байтах.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
        ↪  FileMappedResizableDirectMemory(address, memoryReservationStep),
        ↪  memoryReservationStep) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
        ↪  DefaultLinksSizeStep) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64UnitedMemoryLinks(IResizableDirectMemory memory, long
        ↪  memoryReservationStep) : this(memory, memoryReservationStep,
        ↪  Default<LinksConstants<ulong>>.Instance, true) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64UnitedMemoryLinks(IResizableDirectMemory memory, long
        ↪  memoryReservationStep, LinksConstants<ulong> constants, bool useAvlBasedIndex) :
        ↪  base(memory, memoryReservationStep, constants)
        {
            if (useAvlBasedIndex)
            {
                _createSourceTreeMethods = () => new
                ↪  UInt64LinksSourcesAvlBalancedTreeMethods(Constants, _links, _header);
                _createTargetTreeMethods = () => new
                ↪  UInt64LinksTargetsAvlBalancedTreeMethods(Constants, _links, _header);
            }
            else
            {
                _createSourceTreeMethods = () => new
                ↪  UInt64LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
                _createTargetTreeMethods = () => new
                ↪  UInt64LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
            }
            Init(memory, memoryReservationStep);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetPointers(IResizableDirectMemory memory)
        {
            _header = (LinksHeader<ulong>*)memory.Pointer;
            _links = (RawLink<ulong>*)memory.Pointer;
            SourcesTreeMethods = _createSourceTreeMethods();
            TargetsTreeMethods = _createTargetTreeMethods();
            UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_links, _header);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ResetPointers()
        {
            base.ResetPointers();
            _links = null;
            _header = null;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref LinksHeader<ulong> GetHeaderReference() => ref *_header;
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLink<ulong> GetLinkReference(ulong linkIndex) => ref
         ↪  _links[linkIndex];

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

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

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

        [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 ulong GetZero() => 0UL;

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override long ConvertToInt64(ulong value) => (long)value;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong ConvertToAddress(long value) => (ulong)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;

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

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

1.57 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnusedLinksListMethods.cs

```
using System.Runtime.CompilerServices;
using Platform.Data.Doublets.Memory.United.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Specific
{
    public unsafe class UInt64UnusedLinksListMethods : UnusedLinksListMethods<ulong>
    {
        private readonly RawLink<ulong>* _links;
        private readonly LinksHeader<ulong>* _header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64UnusedLinksListMethods(RawLink<ulong>* links, LinksHeader<ulong>* header)
            : base((byte*)links, (byte*)header)
        {
            _links = links;
            _header = header;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref _links[link];

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref LinksHeader<ulong> GetHeaderReference() => ref *_header;
    }
}
```

1.58 ./csharp/Platform.Data.Doublets/Numbers/Unary/AddressToUnaryNumberConverter.cs

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

```

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Numbers.Unary
{
    public class AddressToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
     ↪  IConverter<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
         ↪  EqualityComparer<TLink>.Default;
        private static readonly TLink _zero = default;
        private static readonly TLink _one = Arithmetic.Increment(_zero);

        private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
         ↪  powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
         ↪  powerOf2ToUnaryNumberConverter;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(TLink number)
        {
            var links = _links;
            var nullConstant = links.Constants.Null;
            var target = nullConstant;
            for (var i = 0; !_equalityComparer.Equals(number, _zero) && i <
             ↪  NumericType<TLink>.BitsSize; i++)
            {
                if (_equalityComparer.Equals(Bit.And(number, _one), _one))
                {
                    target = _equalityComparer.Equals(target, nullConstant)
                        ? _powerOf2ToUnaryNumberConverter.Convert(i)
                        : links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
                }
                number = Bit.ShiftRight(number, 1);
            }
            return target;
        }
    }
}
```

1.59 ./csharp/Platform.Data.Doublets/Numbers/Unary/LinkToItsFrequencyNumberConveter.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkToItsFrequencyNumberConveter(
            ILinks<TLink> links,
            IProperty<TLink, TLink> frequencyPropertyOperator,
            IConverter<TLink> unaryNumberToAddressConverter)
            : base(links)
        {
            _frequencyPropertyOperator = frequencyPropertyOperator;
            _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(Doublet<TLink> doublet)
        {
            var links = _links;
            var link = links.SearchOrDefault(doublet.Source, doublet.Target);
            if (_equalityComparer.Equals(link, default))
            {
                throw new ArgumentException($"Link ({doublet}) not found.", nameof(doublet));
            }
            var frequency = _frequencyPropertyOperator.Get(link);
            if (_equalityComparer.Equals(frequency, default))
            {
                return default;
            }
            var frequencyNumber = links.GetSource(frequency);
            return _unaryNumberToAddressConverter.Convert(frequencyNumber);
        }
    }
}
```

1.60 ./csharp/Platform.Data.Doublets/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs

```
using System.Collections.Generic;
using Platform.Exceptions;
using Platform.Ranges;
using Platform.Converters;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

        private readonly TLink[] _unaryNumberPowersOf2;

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

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

1.61 ./csharp/Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressAddOperationConverter.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Numbers.Unary
{
    public class UnaryNumberToAddressAddOperationConverter<TLink> : LinksOperatorBase<TLink>,
     ↪  IConverter<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
         ↪  EqualityComparer<TLink>.Default;
        private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
         ↪  UncheckedConverter<TLink, ulong>.Default;
        private static readonly UncheckedConverter<ulong, TLink> _uInt64ToAddressConverter =
         ↪  UncheckedConverter<ulong, TLink>.Default;
        private static readonly TLink _zero = default;
        private static readonly TLink _one = Arithmetic.Increment(_zero);

        private readonly Dictionary<TLink, TLink> _unaryToUInt64;
        private readonly TLink _unaryOne;
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnaryNumberToAddressAddOperationConverter(ILinks<TLink> links, TLink unaryOne)
            : base(links)
        {
            _unaryOne = unaryOne;
            _unaryToUInt64 = CreateUnaryToUInt64Dictionary(links, unaryOne);
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Dictionary<TLink, TLink> CreateUnaryToUInt64Dictionary(ILinks<TLink>
        ↪  links, TLink unaryOne)
        {
            var unaryToUInt64 = new Dictionary<TLink, TLink>
            {
                { unaryOne, _one }
            };
            var unary = unaryOne;
            var number = _one;
            for (var i = 1; i < 64; i++)
            {
                unary = links.GetOrCreate(unary, unary);
                number = Double(number);
                unaryToUInt64.Add(unary, number);
            }
            return unaryToUInt64;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLink Double(TLink number) =>
        ↪  _uInt64ToAddressConverter.Convert(_addressToUInt64Converter.Convert(number) * 2UL);
    }
}
```

1.62 ./csharp/Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressOrOperationConverter.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Numbers.Unary
{
    public class UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
         ↪  EqualityComparer<TLink>.Default;
        private static readonly TLink _zero = default;
        private static readonly TLink _one = Arithmetic.Increment(_zero);

        private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
         ↪  TLink> powerOf2ToUnaryNumberConverter) : base(links) => _unaryNumberPowerOf2Indicies
         ↪  = CreateUnaryNumberPowerOf2IndiciesDictionary(powerOf2ToUnaryNumberConverter);

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Dictionary<TLink, int>
         ↪  CreateUnaryNumberPowerOf2IndiciesDictionary(IConverter<int, TLink>
         ↪  powerOf2ToUnaryNumberConverter)
        {
            var unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
            for (int i = 0; i < NumericType<TLink>.BitsSize; i++)
            {
                unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
            }
            return unaryNumberPowerOf2Indicies;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void SetBit(ref TLink target, int powerOf2Index) => target =
         ↪  Bit.Or(target, Bit.ShiftLeft(_one, powerOf2Index));
    }
}
```

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public PropertiesOperator(ILinks<TLink> links) : base(links) { }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void SetValue(TLink @object, TLink property, TLink value)
        {
            var links = _links;
            var objectProperty = links.GetOrCreate(@object, property);
            links.DeleteMany(links.AllIndices(links.Constants.Any, objectProperty));
            links.GetOrCreate(objectProperty, value);
        }
    }
}
```

1.64 ./csharp/Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

        private readonly TLink _propertyMarker;
        private readonly TLink _propertyValueMarker;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public PropertyOperator(ILinks<TLink> links, TLink propertyMarker, TLink
         ↪  propertyValueMarker) : base(links)
        {
            _propertyMarker = propertyMarker;
            _propertyValueMarker = propertyValueMarker;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Get(TLink link)
        {
            var property = _links.SearchOrDefault(link, _propertyMarker);
            return GetValue(GetContainer(property));
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private TLink GetContainer(TLink property)
        {
            var valueContainer = default(TLink);
            if (_equalityComparer.Equals(property, default))
            {
                return valueContainer;
            }
            var links = _links;
            var constants = links.Constants;
            var countinueConstant = constants.Continue;
            var breakConstant = constants.Break;
            var anyConstant = constants.Any;
            var query = new Link<TLink>(anyConstant, property, anyConstant);
            links.Each(candidate =>
            {
                var candidateTarget = links.GetTarget(candidate);
                var valueTarget = links.GetTarget(candidateTarget);
                if (_equalityComparer.Equals(valueTarget, _propertyValueMarker))
                {
                    valueContainer = links.GetIndex(candidate);
                    return breakConstant;
                }
                return countinueConstant;
            }, query);
            return valueContainer;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private TLink GetValue(TLink container) => _equalityComparer.Equals(container, default)
        ↪  ? default : _links.GetTarget(container);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Set(TLink link, TLink value)
        {
            var links = _links;
            var property = links.GetOrCreate(link, _propertyMarker);
            var container = GetContainer(property);
            if (_equalityComparer.Equals(container, default))
            {
                links.GetOrCreate(property, value);
            }
            else
            {
                links.Update(container, property, value);
            }
        }
    }
}
```

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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]);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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];
            }
        }
    }
}
```

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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 LinksConstants<TLink> _constants =
        ↪  Default<LinksConstants<TLink>>.Instance;
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        private static readonly TLink _zero = default;
        private static readonly TLink _one = Arithmetic.Increment(_zero);

        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;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
            {
                Element = element;
                DoubletData = doubletData;
            }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
        ↪  baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
            : this(links, baseConverter, doubletFrequenciesCache, _one, true) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
        ↪  baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
        ↪  doInitialFrequenciesIncrement)
            : this(links, baseConverter, doubletFrequenciesCache, _one,
            ↪  doInitialFrequenciesIncrement) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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, _one) < 0)
            {
                minFrequencyToCompress = _one;
            }
            _minFrequencyToCompress = minFrequencyToCompress;
            _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
            ResetMaxDoublet();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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 &&
               (_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) */
            {
                _maxDoublet = doublet;
                _maxDoubletData = data;
            }
        }
    }
}
```

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Converters
{
    public abstract class LinksListToSequenceConverterBase<TLink> : LinksOperatorBase<TLink>,
     ↪  IConverter<IList<TLink>, TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected LinksListToSequenceConverterBase(ILinks<TLink> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public abstract TLink Convert(IList<TLink> source);
    }
}
```

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

```
using System.Collections.Generic;
using System.Linq;
using System.Runtime.CompilerServices;
using Platform.Converters;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

        private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
         ↪  sequenceToItsLocalElementLevelsConverter) : base(links)
            => _sequenceToItsLocalElementLevelsConverter =
             ↪  sequenceToItsLocalElementLevelsConverter;

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
         ↪  _comparer.Compare(next, current) < 0 ? next : current;
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
        ↪  => _comparer.Compare(previous, current) < 0 ? previous : current;
    }
}
```

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SequenceToItsLocalElementLevelsConverter(ILinks<TLink> links,
        ↪  IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter) : base(links)
        ↪  => _linkToItsFrequencyToNumberConveter = linkToItsFrequencyToNumberConveter;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }

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

1.70 ./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/DefaultSequenceElementCriterionMatcher.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.CriterionMatchers
{
    public class DefaultSequenceElementCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
    ↪  ICriterionMatcher<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public DefaultSequenceElementCriterionMatcher(ILinks<TLink> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool IsMatched(TLink argument) => _links.IsPartialPoint(argument);
    }
}
```

1.71 ./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/MarkedSequenceCriterionMatcher.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public MarkedSequenceCriterionMatcher(ILinks<TLink> links, TLink sequenceMarkerLink)
        {
            _links = links;
            _sequenceMarkerLink = sequenceMarkerLink;
        }

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

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
         ↪  ISequenceHeightProvider<TLink> heightProvider)
            : base(links)
        {
            _stack = stack;
            _heightProvider = heightProvider;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Append(TLink sequence, TLink appendant)
        {
            var cursor = sequence;
            var links = _links;
            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);
        }
        }
    }
```

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences
{
    public class DuplicateSegmentsCounter<TLink> : ICounter<int>
    {
        private readonly IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
            _duplicateFragmentsProvider;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLink>,
            IList<TLink>>>> duplicateFragmentsProvider) => _duplicateFragmentsProvider =
            duplicateFragmentsProvider;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
    }
}
```

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

```
using System;
using System.Linq;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Collections;
using Platform.Collections.Lists;
using Platform.Collections.Segments;
using Platform.Collections.Segments.Walkers;
using Platform.Singletons;
using Platform.Converters;
using Platform.Data.Doublets.Unicode;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences
{
    public class DuplicateSegmentsProvider<TLink> :
        DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
        IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
    {
        private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
            UncheckedConverter<TLink, long>.Default;
        private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
            UncheckedConverter<TLink, ulong>.Default;
        private static readonly UncheckedConverter<ulong, TLink> _uInt64ToAddressConverter =
            UncheckedConverter<ulong, TLink>.Default;

        private readonly ILinks<TLink> _links;
        private readonly ILinks<TLink> _sequences;
        private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
        private BitString _visited;

        private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
            IList<TLink>>>
        {
            private readonly IListEqualityComparer<TLink> _listComparer;

            public ItemEquilityComparer() => _listComparer =
                Default<IListEqualityComparer<TLink>>.Instance;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            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);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
                (_listComparer.GetHashCode(pair.Key),
                _listComparer.GetHashCode(pair.Value)).GetHashCode();
        }
        private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
        {
            private readonly IListComparer<TLink> _listComparer;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
            ↪  KeyValuePair<IList<TLink>, IList<TLink>> right)
            {
                var intermediateResult = _listComparer.Compare(left.Key, right.Key);
                if (intermediateResult == 0)
                {
                    intermediateResult = _listComparer.Compare(left.Value, right.Value);
                }
                return intermediateResult;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public DuplicateSegmentsProvider(ILinks<TLink> links, ILinks<TLink> sequences)
            : base(minimumStringSegmentLength: 2)
        {
            _links = links;
            _sequences = sequences;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
        {
            _groups = new HashSet<KeyValuePair<IList<TLink>,
            ↪  IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
            var links = _links;
            var count = links.Count();
            _visited = new BitString(_addressToInt64Converter.Convert(count) + 1L);
            links.Each(link =>
            {
                var linkIndex = links.GetIndex(link);
                var linkBitIndex = _addressToInt64Converter.Convert(linkIndex);
                var constants = links.Constants;
                if (!_visited.Get(linkBitIndex))
                {
                    var sequenceElements = new List<TLink>();
                    var filler = new ListFiller<TLink, TLink>(sequenceElements, constants.Break);
                    _sequences.Each(filler.AddSkipFirstAndReturnConstant, new
                    ↪  LinkAddress<TLink>(linkIndex));
                    if (sequenceElements.Count > 2)
                    {
                        WalkAll(sequenceElements);
                    }
                }
                return constants.Continue;
            });
            var resultList = _groups.ToList();
            var comparer = Default<ItemComparer>.Instance;
            resultList.Sort(comparer);
#if DEBUG
            foreach (var item in resultList)
            {
                PrintDuplicates(item);
            }
#endif
            return resultList;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
        ↪  length) => new Segment<TLink>(elements, offset, length);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void OnDublicateFound(Segment<TLink> segment)
        {
            var duplicates = CollectDuplicatesForSegment(segment);
            if (duplicates.Count > 1)
            {
                _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
                ↪  duplicates));
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
        {
            var duplicates = new List<TLink>();
            var readAsElement = new HashSet<TLink>();
            var restrictions = segment.ShiftRight();
            var constants = _links.Constants;
            restrictions[0] = constants.Any;
            _sequences.Each(sequence =>
            {
                var sequenceIndex = sequence[constants.IndexPart];
                duplicates.Add(sequenceIndex);
                readAsElement.Add(sequenceIndex);
                return constants.Continue;
            }, restrictions);
            if (duplicates.Any(x => _visited.Get(_addressToInt64Converter.Convert(x))))
            {
                return new List<TLink>();
            }
            foreach (var duplicate in duplicates)
            {
                var duplicateBitIndex = _addressToInt64Converter.Convert(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)
                {
                    var sequenceIndex =
                    ↪  _uInt64ToAddressConverter.Convert(partiallyMatchedSequence);
                    duplicates.Add(sequenceIndex);
                }
            }
            duplicates.Sort();
            return duplicates;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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++)
            {
                var sequenceIndex = _addressToUInt64Converter.Convert(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();
        }
    }
}
```

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

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

```

```csharp
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

        private static readonly TLink _zero = default;
        private static readonly TLink _one = Arithmetic.Increment(_zero);

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
        {
            _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
            return data;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void IncrementFrequencies(IList<TLink> sequence)
        {
            for (var i = 1; i < sequence.Count; i++)
            {
                IncrementFrequency(sequence[i - 1], sequence[i]);
            }
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void PrintFrequencies(IList<TLink> sequence)
        {
            for (var i = 1; i < sequence.Count; i++)
            {
                PrintFrequency(sequence[i - 1], sequence[i]);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void PrintFrequency(TLink source, TLink target)
        {
            var number = GetFrequency(source, target).Frequency;
            Console.WriteLine("({0},{1}) - {2}", source, target, number);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
        {
            if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
            {
                data.IncrementFrequency();
            }
            else
            {
                var link = _links.SearchOrDefault(doublet.Source, doublet.Target);
                data = new LinkFrequency<TLink>(_one, link);
                if (!_equalityComparer.Equals(link, default))
                {
                    data.Frequency = Arithmetic.Add(data.Frequency,
                     ↪  _frequencyCounter.Count(link));
                }
                _doubletsCache.Add(doublet, data);
            }
            return data;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void ValidateFrequencies()
        {
            foreach (var entry in _doubletsCache)
            {
                var value = entry.Value;
                var linkIndex = value.Link;
                if (!_equalityComparer.Equals(linkIndex, default))
                {
                    var frequency = value.Frequency;
                    var count = _frequencyCounter.Count(linkIndex);
                    // TODO: Why `frequency` always greater than `count` by 1?
                    if (((_comparer.Compare(frequency, count) > 0) &&
                     ↪  (_comparer.Compare(Arithmetic.Subtract(frequency, count), _one) > 0))
                     || ((_comparer.Compare(count, frequency) > 0) &&
                      ↪  (_comparer.Compare(Arithmetic.Subtract(count, frequency), _one) > 0)))
                    {
                        throw new InvalidOperationException("Frequencies validation failed.");
                    }
                }
                //else
                //{
                //    if (value.Frequency > 0)
                //    {
                //        var frequency = value.Frequency;
                //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
                //        var count = _countLinkFrequency(linkIndex);

                //        if ((frequency > count && frequency - count > 1) || (count > frequency
                ↪  && count - frequency > 1))
                //            throw new InvalidOperationException("Frequencies validation
                ↪  failed.");
                //    }
                //}
            }
        }
    }
}
```

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency(TLink frequency, TLink link)
        {
            Frequency = frequency;
            Link = link;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency() { }

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

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

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

1.77 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToItsFrequencyValueConverter.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
{
    public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLink> :
     ↪  IConverter<Doublet<TLink>, TLink>
    {
        private readonly LinkFrequenciesCache<TLink> _cache;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public
         ↪  FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink>
         ↪  cache) => _cache = cache;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref source).Frequency;
    }
}
```

1.78 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOf

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
         ↪  ICriterionMatcher<TLink> markedSequenceMatcher, TLink sequenceLink, TLink symbol)
            : base(links, sequenceLink, symbol)
            => _markedSequenceMatcher = markedSequenceMatcher;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Count()
        {
            if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
            {
                return default;
            }
            return base.Count();
        }
    }
}
```

1.79 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCount

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink sequenceLink,
         ↪  TLink symbol)
        {
            _links = links;
            _sequenceLink = sequenceLink;
            _symbol = symbol;
            _total = default;
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool VisitElement(TLink element)
        {
            if (_equalityComparer.Equals(element, _symbol))
            {
                _total = Arithmetic.Increment(_total);
            }
            return true;
        }
    }
}
```

1.80 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyC

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TotalMarkedSequenceSymbolFrequencyCounter(ILinks<TLink> links,
         ↪  ICriterionMatcher<TLink> markedSequenceMatcher)
        {
            _links = links;
            _markedSequenceMatcher = markedSequenceMatcher;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Count(TLink argument) => new
         ↪  TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
         ↪  _markedSequenceMatcher, argument).Count();
    }
}
```

1.81 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyC

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

```csharp

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TotalMarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
         ↪  ICriterionMatcher<TLink> markedSequenceMatcher, TLink symbol)
            : base(links, symbol)
            => _markedSequenceMatcher = markedSequenceMatcher;

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

1.82 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.

```csharp
using System.Runtime.CompilerServices;
using Platform.Interfaces;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TotalSequenceSymbolFrequencyCounter(ILinks<TLink> links) => _links = links;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Count(TLink symbol) => new
         ↪  TotalSequenceSymbolFrequencyOneOffCounter<TLink>(_links, symbol).Count();
    }
}
```

1.83 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffC

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
        {
            _links = links;
            _symbol = symbol;
            _visits = new HashSet<TLink>();
            _total = default;
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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);
            }
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private TLink EachElementHandler(IList<TLink> doublet)
        {
            var constants = _links.Constants;
            var doubletIndex = doublet[constants.IndexPart];
            if (_visits.Add(doubletIndex))
            {
                CountCore(doubletIndex);
            }
            return constants.Continue;
        }
    }
}
```

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.HeightProviders
{
    public class CachedSequenceHeightProvider<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 IProperties<TLink, TLink, TLink> _propertyOperator;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CachedSequenceHeightProvider(
            ISequenceHeightProvider<TLink> baseHeightProvider,
            IConverter<TLink> addressToUnaryNumberConverter,
            IConverter<TLink> unaryNumberToAddressConverter,
            TLink heightPropertyMarker,
            IProperties<TLink, TLink, TLink> propertyOperator)
        {
            _heightPropertyMarker = heightPropertyMarker;
            _baseHeightProvider = baseHeightProvider;
            _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
            _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
            _propertyOperator = propertyOperator;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }
    }
}
```

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriterionMatcher<TLink>
         ↪  elementMatcher) : base(links) => _elementMatcher = elementMatcher;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }
    }
}
```

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

```
using Platform.Interfaces;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

1.87 ./csharp/Platform.Data.Doublets/Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

        private readonly LinkFrequenciesCache<TLink> _cache;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CachedFrequencyIncrementingSequenceIndex(LinkFrequenciesCache<TLink> cache) =>
        ↪  _cache = cache;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Add(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
            ↪  { }
            for (; i >= 1; i--)
            {
                _cache.IncrementFrequency(sequence[i - 1], sequence[i]);
            }
            return indexed;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool IsIndexedWithIncrement(TLink source, TLink target)
        {
            var frequency = _cache.GetFrequency(source, target);
            if (frequency == null)
            {
                return false;
            }
            var indexed = !_equalityComparer.Equals(frequency.Frequency, default);
            if (indexed)
            {
                _cache.IncrementFrequency(source, target);
            }
            return indexed;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool MightContain(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
            return indexed;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool IsIndexed(TLink source, TLink target)
        {
            var frequency = _cache.GetFrequency(source, target);
            if (frequency == null)
            {
                return false;
            }
            return !_equalityComparer.Equals(frequency.Frequency, default);
        }
    }
}
```

1.88 ./csharp/Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public FrequencyIncrementingSequenceIndex(ILinks<TLink> links, IProperty<TLink, TLink>
        ↪  frequencyPropertyOperator, IIncrementer<TLink> frequencyIncrementer)
```

```
            : base(links)
        {
            _frequencyPropertyOperator = frequencyPropertyOperator;
            _frequencyIncrementer = frequencyIncrementer;
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool IsIndexedWithIncrement(TLink source, TLink target)
        {
            var link = _links.SearchOrDefault(source, target);
            var indexed = !_equalityComparer.Equals(link, default);
            if (indexed)
            {
                Increment(link);
            }
            return indexed;
        }

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

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        bool MightContain(IList<TLink> sequence);
    }
}
```

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SequenceIndex(ILinks<TLink> links) : base(links) { }

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

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

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

        private readonly ISynchronizedLinks<TLink> _links;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SynchronizedSequenceIndex(ISynchronizedLinks<TLink> links) => _links = links;

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool MightContain(IList<TLink> sequence)
        {
            var links = _links.Unsync;
            return _links.SyncRoot.ExecuteReadOperation(() =>
            {
                var indexed = true;
                var i = sequence.Count;
                while (--i >= 1 && (indexed =
                 ↪  !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                 ↪  sequence[i]), default))) { }
                return indexed;
            });
        }
    }
}
```

1.92 ./csharp/Platform.Data.Doublets/Sequences/Indexes/Unindex.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Indexes
{
    public class Unindex<TLink> : ISequenceIndex<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool Add(IList<TLink> sequence) => false;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool MightContain(IList<TLink> sequence) => true;
    }
}
```

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

        /// <remarks>
        /// Number of links that is needed to generate all variants for
        /// sequence of length N corresponds to https://oeis.org/A014143/list sequence.
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ulong[] CreateAllVariants2(ulong[] sequence)
        {
            return _sync.ExecuteWriteOperation(() =>
            {
                if (sequence.IsNullOrEmpty())
                {
                    return Array.Empty<ulong>();
                }
                Links.EnsureLinkExists(sequence);
                if (sequence.Length == 1)
                {
                    return sequence;
                }
                return CreateAllVariants2Core(sequence, 0, sequence.Length - 1);
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private ulong[] CreateAllVariants2Core(ulong[] sequence, long startAt, long stopAt)
        {
#if DEBUG
            if ((stopAt - startAt) < 0)
            {
                throw new ArgumentOutOfRangeException(nameof(startAt), "startAt должен быть
                ↪  меньше или равен stopAt");
            }
#endif
            if ((stopAt - startAt) == 0)
            {
                return new[] { sequence[startAt] };
            }
            if ((stopAt - startAt) == 1)
            {
                return new[] { Links.Unsync.GetOrCreate(sequence[startAt], sequence[stopAt]) };
            }
            var variants = new ulong[(ulong)Platform.Numbers.Math.Catalan(stopAt - startAt)];
            var last = 0;
            for (var splitter = startAt; splitter < stopAt; splitter++)
            {
                var left = CreateAllVariants2Core(sequence, startAt, splitter);
                var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
                for (var i = 0; i < left.Length; i++)
                {
                    for (var j = 0; j < right.Length; j++)
                    {
                        var variant = Links.Unsync.GetOrCreate(left[i], right[j]);
                        if (variant == Constants.Null)
                        {
                            throw new NotImplementedException("Creation cancellation is not
                            ↪  implemented.");
                        }
                        variants[last++] = variant;
                    }
                }
            }
            return variants;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<ulong> CreateAllVariants1(params ulong[] sequence)
        {
            return _sync.ExecuteWriteOperation(() =>
            {
                if (sequence.IsNullOrEmpty())
                {
                    return new List<ulong>();
                }
                Links.Unsync.EnsureLinkExists(sequence);
                if (sequence.Length == 1)
                {
                    return new List<ulong> { sequence[0] };
                }
                var results = new
                ↪  List<ulong>((int)Platform.Numbers.Math.Catalan(sequence.Length));
                return CreateAllVariants1Core(sequence, results);
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
        {
            if (sequence.Length == 2)
            {
                var link = Links.Unsync.GetOrCreate(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.GetOrCreate(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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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(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[Constants.IndexPart];
                        Each1(handler, innerSequence);
                        return Constants.Continue;
                    }, Constants.Any, left, right);
                }
            }
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void EachPart(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[] sequence)
        {
            var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
            EachPartCore(link =>
            {
                var linkIndex = link[Constants.IndexPart];
                if (!visitedLinks.Contains(linkIndex))
                {
                    visitedLinks.Add(linkIndex); // изучить почему случаются повторы
                    return handler(new LinkAddress<LinkIndex>(linkIndex));
                }
                return Constants.Continue;
            }, sequence);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void EachPartCore(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[]
        ↪   sequence)
        {
            if (sequence.IsNullOrEmpty())
            {
                return;
            }
            Links.EnsureLinkIsAnyOrExists(sequence);
            if (sequence.Length == 1)
            {
                var link = sequence[0];
                if (link > 0)
                {
                    handler(new LinkAddress<LinkIndex>(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(new LinkAddress<LinkIndex>(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(new LinkAddress<LinkIndex>(match));
                    }
                    return true;
                });
                //          ._x o_.
                //           |___|
                PartialStepRight(x => handler(x), sequence[0], sequence[1]);
            }
            else
            {
                throw new NotImplementedException();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void PartialStepRight(Action<IList<LinkIndex>> 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;
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void StepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
        {
            Links.Unsync.Each(left, Constants.Any, rightStep =>
            {
                TryStepRightUp(handler, right, rightStep);
                return true;
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void TryStepRightUp(Action<IList<LinkIndex>> 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(new LinkAddress<LinkIndex>(stepFrom));
            }
        }

        // TODO: Test
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void PartialStepLeft(Action<IList<LinkIndex>> 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;
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
        {
            Links.Unsync.Each(Constants.Any, right, leftStep =>
            {
                TryStepLeftUp(handler, left, leftStep);
                return true;
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void TryStepLeftUp(Action<IList<LinkIndex>> handler, ulong left, ulong stepFrom)
        {
            var upStep = stepFrom;
            var firstTarget = Links.Unsync.GetSource(upStep);
            while (firstTarget != left && firstTarget != upStep)
            {
                upStep = firstTarget;
                firstTarget = Links.Unsync.GetTarget(upStep);
            }
            if (firstTarget == left)
            {
                handler(new LinkAddress<LinkIndex>(stepFrom));
            }
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var results = new List<ulong>();
                if (sequence.Length > 0)
                {
                    Links.EnsureLinkExists(sequence);
                    var firstElement = sequence[0];
                    if (sequence.Length == 1)
                    {
                        results.Add(firstElement);
                        return results;
                    }
                    if (sequence.Length == 2)
                    {
                        var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
                        if (doublet != Constants.Null)
                        {
                            results.Add(doublet);
                        }
                        return results;
                    }
                    var linksInSequence = new HashSet<ulong>(sequence);
                    void handler(IList<LinkIndex> result)
                    {
                        var resultIndex = result[Links.Constants.IndexPart];
                        var filterPosition = 0;
                        StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
                         ↪  Links.Unsync.GetTarget,
                            x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
                             ↪  x =>
                            {
                                if (filterPosition == sequence.Length)
                                {
                                    filterPosition = -2; // Длиннее чем нужно
                                    return false;
                                }
                                if (x != sequence[filterPosition])
                                {
                                    filterPosition = -1;
                                    return false; // Начинается иначе
                                }
                                filterPosition++;

                                return true;
                            });
                        if (filterPosition == sequence.Length)
                        {
                            results.Add(resultIndex);
                        }
                    }
                    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;
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var results = new HashSet<ulong>();
                if (sequence.Length > 0)
                {
                    Links.EnsureLinkExists(sequence);
                    var firstElement = sequence[0];
                    if (sequence.Length == 1)
                    {
                        results.Add(firstElement);
                        return results;
                    }
                    if (sequence.Length == 2)
                    {
                        var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
                        if (doublet != Constants.Null)
                        {
                            results.Add(doublet);
                        }
                        return results;
                    }
                    var matcher = new Matcher(this, sequence, results, null);
                    if (sequence.Length >= 2)
                    {
                        StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
                    }
                    var last = sequence.Length - 2;
                    for (var i = 1; i < last; i++)
                    {
                        PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
                         ↪  sequence[i + 1]);
                    }
                    if (sequence.Length >= 3)
                    {
                        StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
                         ↪  sequence[sequence.Length - 1]);
                    }
                }
                return results;
            });
        }

        public const int MaxSequenceFormatSize = 200;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
         ↪  => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) => Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink, elementToString, insertComma, knownElements));

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink, Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements)
        {
            var linksInSequence = new HashSet<ulong>(knownElements);
            var entered = new HashSet<ulong>();
            var sb = new StringBuilder();
            sb.Append('{');
            if (links.Exists(sequenceLink))
            {
                StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
                    x => linksInSequence.Contains(x) || links.IsFullPoint(x), entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
                    {
                        if (insertComma && sb.Length > 1)
                        {
                            sb.Append(',');
                        }
                        if (entered.Contains(element))
```

```csharp
                        {
                            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();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                if (sequence.Length > 0)
                {
                    Links.EnsureLinkExists(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>();
            });
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                if (sequence.Length > 0)
                {
                    Links.EnsureLinkExists(sequence);
                    var results = new HashSet<ulong>();
                    for (var i = 0; i < sequence.Length; i++)
                    {
                        AllUsagesCore(sequence[i], results);
                    }
                    var filteredResults = new HashSet<ulong>();
                    var matcher = new Matcher(this, sequence, filteredResults, null);
                    matcher.AddAllPartialMatchedToResults(results);
                    return filteredResults;
                }
                return new HashSet<ulong>();
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool GetAllPartiallyMatchingSequences2(Func<IList<LinkIndex>, LinkIndex> handler,
         ↪  params ulong[] sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                if (sequence.Length > 0)
                {
                    Links.EnsureLinkExists(sequence);

                    var results = new HashSet<ulong>();
                    var filteredResults = new HashSet<ulong>();
                    var matcher = new Matcher(this, sequence, filteredResults, handler);
                    for (var i = 0; i < sequence.Length; i++)
                    {
                        if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
                        {
                            return false;
                        }
                    }
                    return true;
                }
                return true;
            });
        }

        //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
        //{
        //    return Sync.ExecuteReadOperation(() =>
        //    {
        //        if (sequence.Length > 0)
        //        {
        //            _links.EnsureEachLinkIsAnyOrExists(sequence);

        //            var firstResults = new HashSet<ulong>();
        //            var lastResults = new HashSet<ulong>();

        //            var first = sequence.First(x => x != LinksConstants.Any);
        //            var last = sequence.Last(x => x != LinksConstants.Any);

        //            AllUsagesCore(first, firstResults);
        //            AllUsagesCore(last, lastResults);

        //            firstResults.IntersectWith(lastResults);

        //            //for (var i = 0; i < sequence.Length; i++)
        //            //    AllUsagesCore(sequence[i], results);

        //            var filteredResults = new HashSet<ulong>();
        //            var matcher = new Matcher(this, sequence, filteredResults, null);
        //            matcher.AddAllPartialMatchedToResults(firstResults);
        //            return filteredResults;
        //        }

        //        return new HashSet<ulong>();
        //    });
        //}
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
        {
            return _sync.ExecuteReadOperation((Func<HashSet<ulong>>)(() =>
            {
                if (sequence.Length > 0)
                {
                    ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links,
                     ↪  (IList<ulong>)sequence);
                    var firstResults = new HashSet<ulong>();
                    var lastResults = new HashSet<ulong>();
                    var first = sequence.First(x => x != Constants.Any);
                    var last = sequence.Last(x => x != Constants.Any);
                    AllUsagesCore(first, firstResults);
                    AllUsagesCore(last, lastResults);
                    firstResults.IntersectWith(lastResults);
                    //for (var i = 0; i < sequence.Length; i++)
                    //    AllUsagesCore(sequence[i], results);
                    var filteredResults = new HashSet<ulong>();
                    var matcher = new Matcher(this, sequence, filteredResults, null);
                    matcher.AddAllPartialMatchedToResults(firstResults);
                    return filteredResults;
                }
                return new HashSet<ulong>();
            }));
        }

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> AllBottomUsages(ulong link)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var visits = new HashSet<ulong>();
                var usages = new HashSet<ulong>();
                AllBottomUsagesCore(link, visits, usages);
                return usages;
            });
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<IList<LinkIndex>, LinkIndex> outerHandler)
        {
            bool handler(ulong doublet)
            {
                if (usages.Add(doublet))
                {
                    if (outerHandler(new LinkAddress<LinkIndex>(doublet)) != Constants.Continue)
                    {
                        return false;
                    }
                    if (!AllUsagesCore1(doublet, usages, outerHandler))
                    {
                        return false;
                    }
                }
                return true;
            }
            return Links.Unsync.Each(link, Constants.Any, handler)
                && Links.Unsync.Each(Constants.Any, link, handler);
        }

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

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

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

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
            ↪  CalculateCore);

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

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

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
            ↪  CalculateCore);

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

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

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
            {
                _links = links;
                _usages = usages;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public bool Collect(ulong link)
            {
                if (_usages.Add(link))
                {
                    _links.Each(link, _links.Constants.Any, Collect);
                    _links.Each(_links.Constants.Any, link, Collect);
                }
                return true;
            }
        }

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

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public ulong Collect(IList<ulong> link)
            {
                var linkIndex = _links.GetIndex(link);
                if (_usages.Add(linkIndex))
                {
                    _links.Each(Collect, _links.Constants.Any, linkIndex);
                }
                return _continue;
            }
        }

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
            {
                _links = links;
                _usages = usages;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public bool Collect(ulong link)
            {
                if (_usages.Add((long)link))
                {
                    _links.Each(link, _links.Constants.Any, Collect);
                    _links.Each(_links.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;

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public bool Collect(ulong link)
            {
                if (_enter.Add(link))
                {
                    if (_intersectWith.Contains(link))
                    {
                        _usages.Add(link);
                    }
                    _links.Unsync.Each(link, _links.Constants.Any, Collect);
                    _links.Unsync.Each(_links.Constants.Any, link, Collect);
                }
                return true;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void CloseInnerConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
         ↪  right)
        {
            TryStepLeftUp(handler, left, right);
            TryStepRightUp(handler, right, left);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void AllCloseConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
         ↪  right)
        {
            // Direct
            if (left == right)
            {
                handler(new LinkAddress<LinkIndex>(left));
            }
            var doublet = Links.Unsync.SearchOrDefault(left, right);
            if (doublet != Constants.Null)
            {
                handler(new LinkAddress<LinkIndex>(doublet));
            }
            // Inner
            CloseInnerConnections(handler, left, right);
            // Outer
            StepLeft(handler, left, right);
            StepRight(handler, left, right);
            PartialStepRight(handler, left, right);
            PartialStepLeft(handler, left, right);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
         ↪  HashSet<ulong> previousMatchings, long startAt)
        {
            if (startAt >= sequence.Length) // ?
            {
                return previousMatchings;
            }
            var secondLinkUsages = new HashSet<ulong>();
            AllUsagesCore(sequence[startAt], secondLinkUsages);
            secondLinkUsages.Add(sequence[startAt]);
            var matchings = new HashSet<ulong>();
            var filler = new SetFiller<LinkIndex, LinkIndex>(matchings, Constants.Continue);
            //for (var i = 0; i < previousMatchings.Count; i++)
            foreach (var secondLinkUsage in secondLinkUsages)
            {
                foreach (var previousMatching in previousMatchings)
                {
                    //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
                     ↪  secondLinkUsage);
                    StepRight(filler.AddFirstAndReturnConstant, previousMatching,
                     ↪  secondLinkUsage);
                    TryStepRightUp(filler.AddFirstAndReturnConstant, secondLinkUsage,
                     ↪  previousMatching);
                    //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
                     ↪  sequence[startAt]); // почему-то эта ошибочная запись приводит к
                     ↪  желаемым результам.
                    PartialStepRight(filler.AddFirstAndReturnConstant, previousMatching,
                     ↪  secondLinkUsage);
                }
            }
            if (matchings.Count == 0)
            {
                return matchings;
            }
            return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
        }

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

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

        // Найти все возможные связи между указанным списком связей.
        // Находит связи между всеми указанными связями в любом порядке.
        // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
         ↪  несколько раз в последовательности)
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var results = new HashSet<ulong>();
                if (linksToConnect.Length > 0)
                {
                    Links.EnsureLinkExists(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;
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var results = new HashSet<ulong>();
                if (linksToConnect.Length > 0)
                {
                    Links.EnsureLinkExists(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;
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var results = new HashSet<ulong>();
                if (linksToConnect.Length > 0)
                {
                    Links.EnsureLinkExists(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;
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var results = new BitString((long)Links.Unsync.Count() + 1); // new BitArray((int)_links.Total + 1);
                if (linksToConnect.Length > 0)
                {
                    Links.EnsureLinkExists(linksToConnect);
                    var collector1 = new AllUsagesCollector2(Links.Unsync, results);
                    collector1.Collect(linksToConnect[0]);
                    for (var i = 1; i < linksToConnect.Length; i++)
                    {
                        var next = new BitString((long)Links.Unsync.Count() + 1); //new BitArray((int)_links.Total + 1);
                        var collector = new AllUsagesCollector2(Links.Unsync, next);
                        collector.Collect(linksToConnect[i]);
                        results = results.And(next);
                    }
                }
                return results.GetSetUInt64Indices();
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static ulong[] Simplify(ulong[] sequence)
        {
            // Считаем новый размер последовательности
            long newLength = 0;
            var zeroOrManyStepped = false;
            for (var i = 0; i < sequence.Length; i++)
            {
                if (sequence[i] == ZeroOrMany)
                {
                    if (zeroOrManyStepped)
                    {
                        continue;
                    }
                    zeroOrManyStepped = true;
                }
                else
                {
                    //if (zeroOrManyStepped) Is it efficient?
                    zeroOrManyStepped = false;
                }
                newLength++;
            }
            // Строим новую последовательность
            zeroOrManyStepped = false;
            var newSequence = new ulong[newLength];
            long j = 0;
            for (var i = 0; i < sequence.Length; i++)
            {
                //var current = zeroOrManyStepped;
                //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
                //if (current && zeroOrManyStepped)
                //    continue;
                //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
                //if (zeroOrManyStepped && newZeroOrManyStepped)
                //    continue;
                //zeroOrManyStepped = newZeroOrManyStepped;
                if (sequence[i] == ZeroOrMany)
                {
                    if (zeroOrManyStepped)
                    {
                        continue;
                    }
                    zeroOrManyStepped = true;
                }
                else
                {
                    //if (zeroOrManyStepped) Is it efficient?
                    zeroOrManyStepped = false;
                }
                newSequence[j++] = sequence[i];
            }
            return newSequence;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void TestSimplify()
        {
            var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
            var simplifiedSequence = Simplify(sequence);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<ulong> GetSimilarSequences() => new List<ulong>();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Prediction()
        {
            //_links
            //sequences
        }

        #region From Triplets

        //public static void DeleteSequence(Link sequence)
        //{
        //}

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<ulong> CollectMatchingSequences(ulong[] links)
        {
            if (links.Length == 1)
            {
                throw new InvalidOperationException("Подпоследовательности с одним элементом не поддерживаются.");
            }
            var leftBound = 0;
            var rightBound = links.Length - 1;
            var left = links[leftBound++];
            var right = links[rightBound--];
            var results = new List<ulong>();
            CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
            return results;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
         ↪  middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
        {
            var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
            var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
            if (leftLinkTotalReferers <= rightLinkTotalReferers)
            {
                var nextLeftLink = middleLinks[leftBound];
                var elements = GetRightElements(leftLink, nextLeftLink);
                if (leftBound <= rightBound)
                {
                    for (var i = elements.Length - 1; i >= 0; i--)
                    {
                        var element = elements[i];
                        if (element != 0)
                        {
                            CollectMatchingSequences(element, leftBound + 1, middleLinks,
                             ↪  rightLink, rightBound, ref results);
                        }
                    }
                }
                else
                {
                    for (var i = elements.Length - 1; i >= 0; i--)
                    {
                        var element = elements[i];
                        if (element != 0)
                        {
                            results.Add(element);
                        }
                    }
                }
            }
            else
            {
                var nextRightLink = middleLinks[rightBound];
                var elements = GetLeftElements(rightLink, nextRightLink);
                if (leftBound <= rightBound)
                {
                    for (var i = elements.Length - 1; i >= 0; i--)
                    {
                        var element = elements[i];
                        if (element != 0)
                        {
                            CollectMatchingSequences(leftLink, leftBound, middleLinks,
                             ↪  elements[i], rightBound - 1, ref results);
                        }
                    }
                }
                else
                {
                    for (var i = elements.Length - 1; i >= 0; i--)
                    {
                        var element = elements[i];
                        if (element != 0)
                        {
                            results.Add(element);
                        }
                    }
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }

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

        #endregion

        #region Walkers

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

            #region Pattern Match

            enum PatternBlockType
            {
                Undefined,
                Gap,
                Elements
            }

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

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

            #endregion

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
            ↪  base.IsElement(link);

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

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

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

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

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

            //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out
            ↪  long maximumGap)
            //{
            //    mininumGap = 0;
            //    maximumGap = 0;
            //    element = 0;
            //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
            //    {
            //        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
            //            mininumGap++;
            //        else if (_patternSequence[_patternPosition] == ZeroOrMany)
            //            maximumGap = long.MaxValue;
            //        else
            //            break;
            //    }

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

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

```
                    //    return false;
                    //if (_filterPosition >= 0)
                    //{
                    //    if (element == _patternSequence[_filterPosition + 1])
                    //        _filterPosition++;
                    //    else
                    //        return false;
                    //}
                    //if (_filterPosition < 0)
                    //{
                    //    if (element == _patternSequence[0])
                    //        _filterPosition = 0;
                    //}
                }

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

            #endregion
        }
    }
```

1.94 ./csharp/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.Collections.Stacks;
using Platform.Threading.Synchronization;
using Platform.Data.Doublets.Sequences.Walkers;
using LinkIndex = System.UInt64;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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 : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
     ↪  (после завершения реализации Sequences)
    {
        /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
         ↪  связей.</summary>
        public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;

        public SequencesOptions<LinkIndex> Options { get; }
        public SynchronizedLinks<LinkIndex> Links { get; }
        private readonly ISynchronization _sync;

        public LinksConstants<LinkIndex> Constants { get; }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
        {
            Links = links;
            _sync = links.SyncRoot;
            Options = options;
            Options.ValidateOptions();
            Options.InitOptions(Links);
            Constants = links.Constants;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Sequences(SynchronizedLinks<LinkIndex> links) : this(links, new
         ↪  SequencesOptions<LinkIndex>()) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool IsSequence(LinkIndex sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                if (Options.UseSequenceMarker)
                {
                    return Options.MarkedSequenceMatcher.IsMatched(sequence);
                }
                return !Links.Unsync.IsPartialPoint(sequence);
            });
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private LinkIndex GetSequenceElements(LinkIndex sequence)
        {
            if (Options.UseSequenceMarker)
            {
                var linkContents = new Link<ulong>(Links.GetLink(sequence));
                if (linkContents.Source == Options.SequenceMarkerLink)
                {
                    return linkContents.Target;
                }
                if (linkContents.Target == Options.SequenceMarkerLink)
                {
                    return linkContents.Source;
                }
            }
            return sequence;
        }

        #region Count

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkIndex Count(IList<LinkIndex> restrictions)
        {
            if (restrictions.IsNullOrEmpty())
            {
                return Links.Count(Constants.Any, Options.SequenceMarkerLink, Constants.Any);
            }
            if (restrictions.Count == 1) // Первая связь это адрес
            {
                var sequenceIndex = restrictions[0];
                if (sequenceIndex == Constants.Null)
                {
                    return 0;
                }
                if (sequenceIndex == Constants.Any)
                {
                    return Count(null);
                }
                if (Options.UseSequenceMarker)
                {
                    return Links.Count(Constants.Any, Options.SequenceMarkerLink, sequenceIndex);
                }
                return Links.Exists(sequenceIndex) ? 1UL : 0;
            }
            throw new NotImplementedException();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private LinkIndex CountUsages(params LinkIndex[] restrictions)
        {
            if (restrictions.Length == 0)
            {
                return 0;
            }
            if (restrictions.Length == 1) // Первая связь это адрес
            {
                if (restrictions[0] == Constants.Null)
                {
                    return 0;
                }
                var any = Constants.Any;
                if (Options.UseSequenceMarker)
                {
                    var elementsLink = GetSequenceElements(restrictions[0]);
                    var sequenceLink = GetSequenceByElements(elementsLink);
                    if (sequenceLink != Constants.Null)
                    {
                        return Links.Count(any, sequenceLink) + Links.Count(any, elementsLink) -
                         ↪  1;
                    }
                    return Links.Count(any, elementsLink);
                }
                return Links.Count(any, restrictions[0]);
            }
            throw new NotImplementedException();
        }

        #endregion

        #region Create

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkIndex Create(IList<LinkIndex> restrictions)
        {
            return _sync.ExecuteWriteOperation(() =>
            {
                if (restrictions.IsNullOrEmpty())
                {
                    return Constants.Null;
                }
                Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
                return CreateCore(restrictions);
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private LinkIndex CreateCore(IList<LinkIndex> restrictions)
        {
            LinkIndex[] sequence = restrictions.SkipFirst();
            if (Options.UseIndex)
            {
                Options.Index.Add(sequence);
            }
            var sequenceRoot = default(LinkIndex);
            if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
            {
                var matches = Each(restrictions);
                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)
            {
                return Links.Unsync.GetOrCreate(Options.SequenceMarkerLink, sequenceRoot);
            }
            return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
        }

        #endregion

        #region Each

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<LinkIndex> Each(IList<LinkIndex> sequence)
        {
            var results = new List<LinkIndex>();
            var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
            Each(filler.AddFirstAndReturnConstant, sequence);
            return results;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkIndex Each(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
         ↪  restrictions)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                if (restrictions.IsNullOrEmpty())
                {
                    return Constants.Continue;
                }
                Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
                if (restrictions.Count == 1)
                {
                    var link = restrictions[0];
                    var any = Constants.Any;
                    if (link == any)
                    {
                        if (Options.UseSequenceMarker)
                        {
                            return Links.Unsync.Each(handler, new Link<LinkIndex>(any,
                             ↪  Options.SequenceMarkerLink, any));
                        }
                        else
                        {
                            return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
                             ↪  any));
                        }
                    }
                    if (Options.UseSequenceMarker)
                    {
                        var sequenceLinkValues = Links.Unsync.GetLink(link);
                        if (sequenceLinkValues[Constants.SourcePart] ==
                         ↪  Options.SequenceMarkerLink)
                        {
                            link = sequenceLinkValues[Constants.TargetPart];
                        }
                    }
                    var sequence = Options.Walker.Walk(link).ToArray().ShiftRight();
                    sequence[0] = link;
                    return handler(sequence);
                }
                else if (restrictions.Count == 2)
                {
                    throw new NotImplementedException();
                }
                else if (restrictions.Count == 3)
                {
                    return Links.Unsync.Each(handler, restrictions);
                }
                else
                {
                    var sequence = restrictions.SkipFirst();
                    if (Options.UseIndex && !Options.Index.MightContain(sequence))
                    {
                        return Constants.Break;
                    }
                    return EachCore(handler, sequence);
                }
            });
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left, LinkIndex right)
        {
            return Links.Unsync.Each(doublet =>
            {
                var doubletIndex = doublet[Constants.IndexPart];
                if (StepRight(handler, doubletIndex, right) != Constants.Continue)
                {
                    return Constants.Break;
                }
                if (left != doubletIndex)
                {
                    return PartialStepRight(handler, doubletIndex, right);
                }
                return Constants.Continue;
            }, new Link<LinkIndex>(Constants.Any, Constants.Any, left));
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private LinkIndex StepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
         ↪  LinkIndex right) => Links.Unsync.Each(rightStep => TryStepRightUp(handler, right,
         ↪  rightStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, left,
         ↪  Constants.Any));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
         ↪  right, LinkIndex 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(new LinkAddress<LinkIndex>(stepFrom));
            }
            return Constants.Continue;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private LinkIndex StepLeft(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
         ↪  LinkIndex right) => Links.Unsync.Each(leftStep => TryStepLeftUp(handler, left,
         ↪  leftStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, Constants.Any,
         ↪  right));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
         ↪  left, LinkIndex 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(new LinkAddress<LinkIndex>(stepFrom));
            }
            return Constants.Continue;
        }

        #endregion

        #region Update

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkIndex Update(IList<LinkIndex> restrictions, IList<LinkIndex> substitution)
        {
            var sequence = restrictions.SkipFirst();
            var newSequence = substitution.SkipFirst();
            if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
            {
                return Constants.Null;
            }
            if (sequence.IsNullOrEmpty())
            {
                return Create(substitution);
            }
            if (newSequence.IsNullOrEmpty())
            {
                Delete(restrictions);
                return Constants.Null;
            }
            return _sync.ExecuteWriteOperation((Func<ulong>)(() =>
            {
                ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links, (IList<ulong>)sequence);
                Links.EnsureLinkExists(newSequence);
                return UpdateCore(sequence, newSequence);
            }));
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
        {
            if (Options.UseGarbageCollection)
            {
                var sequenceElements = GetSequenceElements(sequence);
                var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
                var sequenceLink = GetSequenceByElements(sequenceElements);
                var newSequenceElements = GetSequenceElements(newSequence);
                var newSequenceLink = GetSequenceByElements(newSequenceElements);
                if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
                {
                    if (sequenceLink != Constants.Null)
                    {
                        Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
                    }
                    Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
                }
                ClearGarbage(sequenceElementsContents.Source);
                ClearGarbage(sequenceElementsContents.Target);
            }
            else
            {
                if (Options.UseSequenceMarker)
                {
                    var sequenceElements = GetSequenceElements(sequence);
                    var sequenceLink = GetSequenceByElements(sequenceElements);
                    var newSequenceElements = GetSequenceElements(newSequence);
                    var newSequenceLink = GetSequenceByElements(newSequenceElements);
                    if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
                    {
                        if (sequenceLink != Constants.Null)
                        {
                            Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
                        }
                        Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
                    }
                }
                else
                {
                    if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
                    {
                        Links.Unsync.MergeAndDelete(sequence, newSequence);
                    }
                }
            }
        }

        #endregion

        #region Delete
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Delete(IList<LinkIndex> restrictions)
        {
            _sync.ExecuteWriteOperation(() =>
            {
                var sequence = restrictions.SkipFirst();
                // TODO: Check all options only ones before loop execution
                foreach (var linkToDelete in Each(sequence))
                {
                    DeleteOneCore(linkToDelete);
                }
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void DeleteOneCore(LinkIndex link)
        {
            if (Options.UseGarbageCollection)
            {
                var sequenceElements = GetSequenceElements(link);
                var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
                var sequenceLink = GetSequenceByElements(sequenceElements);
                if (Options.UseCascadeDelete || CountUsages(link) == 0)
                {
                    if (sequenceLink != Constants.Null)
                    {
                        Links.Unsync.Delete(sequenceLink);
                    }
                    Links.Unsync.Delete(link);
                }
                ClearGarbage(sequenceElementsContents.Source);
                ClearGarbage(sequenceElementsContents.Target);
            }
            else
            {
                if (Options.UseSequenceMarker)
                {
                    var sequenceElements = GetSequenceElements(link);
                    var sequenceLink = GetSequenceByElements(sequenceElements);
                    if (Options.UseCascadeDelete || CountUsages(link) == 0)
                    {
                        if (sequenceLink != Constants.Null)
                        {
                            Links.Unsync.Delete(sequenceLink);
                        }
                        Links.Unsync.Delete(link);
                    }
                }
                else
                {
                    if (Options.UseCascadeDelete || CountUsages(link) == 0)
                    {
                        Links.Unsync.Delete(link);
                    }
                }
            }
        }

        #endregion

        #region Compactification

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void CompactAll()
        {
            _sync.ExecuteWriteOperation(() =>
            {
                var sequences = Each((LinkAddress<LinkIndex>)Constants.Any);
                for (int i = 0; i < sequences.Count; i++)
                {
                    var sequence = this.ToList(sequences[i]);
                    Compact(sequence.ShiftRight());
                }
            });
        }

        /// <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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkIndex Compact(IList<LinkIndex> sequence)
        {
            return _sync.ExecuteWriteOperation(() =>
            {
                if (sequence.IsNullOrEmpty())
                {
                    return Constants.Null;
                }
                Links.EnsureInnerReferenceExists(sequence, nameof(sequence));
                return CompactCore(sequence);
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private LinkIndex CompactCore(IList<LinkIndex> sequence) => UpdateCore(sequence,
        ↪  sequence);

        #endregion

        #region Garbage Collection

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void ClearGarbage(LinkIndex link)
        {
            if (IsGarbage(link))
            {
                var contents = new Link<ulong>(Links.GetLink(link));
                Links.Unsync.Delete(link);
                ClearGarbage(contents.Source);
                ClearGarbage(contents.Target);
            }
        }

        #endregion

        #region Walkers

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var links = Links.Unsync;
                foreach (var part in Options.Walker.Walk(sequence))
                {
                    if (!handler(part))
                    {
                        return false;
                    }
                }
                return true;
            });
        }

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

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
             ↪  (_readAsElements != null && _readAsElements.Contains(link)) ||
             ↪  _linksInSequence.Contains(link);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public bool FullMatch(LinkIndex sequenceToMatch)
            {
                _filterPosition = 0;
                foreach (var part in Walk(sequenceToMatch))
                {
                    if (!FullMatchCore(part))
                    {
                        break;
                    }
                }
                return _filterPosition == _patternSequence.Count;
            }

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void AddFullMatchedToResults(IList<LinkIndex> restrictions)
            {
                var sequenceToMatch = restrictions[_links.Constants.IndexPart];
                if (FullMatch(sequenceToMatch))
                {
                    _results.Add(sequenceToMatch);
                }
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public LinkIndex HandleFullMatched(IList<LinkIndex> restrictions)
            {
                var sequenceToMatch = restrictions[_links.Constants.IndexPart];
                if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
                {
                    return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
                }
                return _links.Constants.Continue;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public LinkIndex HandleFullMatchedSequence(IList<LinkIndex> restrictions)
            {
                var sequenceToMatch = restrictions[_links.Constants.IndexPart];
                var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
                if (sequence != _links.Constants.Null && FullMatch(sequenceToMatch) &&
                 ↪  _results.Add(sequenceToMatch))
                {
                    return _stopableHandler(new LinkAddress<LinkIndex>(sequence));
                }
                return _links.Constants.Continue;
            }

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private bool PartialMatchCore(LinkIndex element)
            {
                if (_filterPosition == (_patternSequence.Count - 1))
                {
                    return false; // Нашлось
                }
                if (_filterPosition >= 0)
                {
                    if (element == _patternSequence[_filterPosition + 1])
                    {
                        _filterPosition++;
                    }
                    else
                    {
                        _filterPosition = -1;
                    }
                }
                if (_filterPosition < 0)
                {
                    if (element == _patternSequence[0])
                    {
                        _filterPosition = 0;
                    }
                }
                return true; // Ищем дальше
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void AddPartialMatchedToResults(LinkIndex sequenceToMatch)
            {
                if (PartialMatch(sequenceToMatch))
                {
                    _results.Add(sequenceToMatch);
                }
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public LinkIndex HandlePartialMatched(IList<LinkIndex> restrictions)
            {
                var sequenceToMatch = restrictions[_links.Constants.IndexPart];
                if (PartialMatch(sequenceToMatch))
                {
                    return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
                }
                return _links.Constants.Continue;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void AddAllPartialMatchedToResults(IEnumerable<LinkIndex> sequencesToMatch)
            {
                foreach (var sequenceToMatch in sequencesToMatch)
                {
                    if (PartialMatch(sequenceToMatch))
                    {
                        _results.Add(sequenceToMatch);
                    }
                }
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<LinkIndex>
             ↪  sequencesToMatch)
            {
                foreach (var sequenceToMatch in sequencesToMatch)
                {
                    if (PartialMatch(sequenceToMatch))
                    {
                        _readAsElements.Add(sequenceToMatch);
                        _results.Add(sequenceToMatch);
                    }
                }
            }
        }

        #endregion
    }
}
```

1.95 ./csharp/Platform.Data.Doublets/Sequences/SequencesExtensions.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences
{
    public static class SequencesExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLink Create<TLink>(this ILinks<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.ShiftRight());
            }
            return sequences.Create(finalSequence.ShiftRight());
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IList<TLink> ToList<TLink>(this ILinks<TLink> sequences, TLink sequence)
        {
            var list = new List<TLink>();
            var filler = new ListFiller<TLink, TLink>(list, sequences.Constants.Break);
            sequences.Each(filler.AddSkipFirstAndReturnConstant, new
             ↪  LinkAddress<TLink>(sequence));
            return list;
        }
    }
}
```

1.96 ./csharp/Platform.Data.Doublets/Sequences/SequencesOptions.cs

```
using System;
using System.Collections.Generic;
using Platform.Interfaces;
using Platform.Collections.Stacks;
using Platform.Converters;
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.Walkers;
using Platform.Data.Doublets.Sequences.Indexes;
using Platform.Data.Doublets.Sequences.CriterionMatchers;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public bool UseCascadeUpdate
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public bool UseCascadeDelete
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public bool UseIndex
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        } // TODO: Update Index on sequence update/delete.

        public bool UseSequenceMarker
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public bool UseCompression
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public bool UseGarbageCollection
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public bool EnforceSingleSequenceVersionOnWriteBasedOnNew
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public MarkedSequenceCriterionMatcher<TLink> MarkedSequenceMatcher
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
                set;
            }

            public IConverter<IList<TLink>, TLink> LinksToSequenceConverter
            {
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                get;
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                set;
            }

            public ISequenceIndex<TLink> Index
            {
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                get;
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                set;
            }

            public ISequenceWalker<TLink> Walker
            {
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                get;
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                set;
            }

            public bool ReadFullSequence
            {
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                get;
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                set;
            }

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void InitOptions(ISynchronizedLinks<TLink> links)
            {
                if (UseSequenceMarker)
                {
                    if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
                    {
                        SequenceMarkerLink = links.CreatePoint();
                    }
                    else
                    {
                        if (!links.Exists(SequenceMarkerLink))
                        {
                            var link = links.CreatePoint();
                            if (!_equalityComparer.Equals(link, SequenceMarkerLink))
                            {
                                throw new InvalidOperationException("Cannot recreate sequence marker
                                 ↪  link.");
                            }
                        }
                    }
                    if (MarkedSequenceMatcher == null)
                    {
                        MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
                         ↪  SequenceMarkerLink);
                    }
                }
                var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
                if (UseCompression)
                {
                    if (LinksToSequenceConverter == null)
                    {
                        ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
                        if (UseSequenceMarker)
                        {
                            totalSequenceSymbolFrequencyCounter = new
                             ↪  TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
                             ↪  MarkedSequenceMatcher);
                        }
                    else
                    {
                        totalSequenceSymbolFrequencyCounter = new
                         ↪  TotalSequenceSymbolFrequencyCounter<TLink>(links);
                    }
                    var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
                     ↪  totalSequenceSymbolFrequencyCounter);
                    var compressingConverter = new CompressingConverter<TLink>(links,
                     ↪  balancedVariantConverter, doubletFrequenciesCache);
                    LinksToSequenceConverter = compressingConverter;
                }
            }
            else
            {
                if (LinksToSequenceConverter == null)
                {
                    LinksToSequenceConverter = balancedVariantConverter;
                }
            }
            if (UseIndex && Index == null)
            {
                Index = new SequenceIndex<TLink>(links);
            }
            if (Walker == null)
            {
                Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
            }
        }

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

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Walkers
{
    public interface ISequenceWalker<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        IEnumerable<TLink> Walk(TLink sequence);
    }
}
```

1.98 ./csharp/Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Walkers
{
    public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
         ↪  isElement) : base(links, stack, isElement) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links, stack,
         ↪  links.IsPartialPoint) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetNextElementAfterPop(TLink element) =>
         ↪  _links.GetSource(element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetNextElementAfterPush(TLink element) =>
        ↪  _links.GetTarget(element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override IEnumerable<TLink> WalkContents(TLink element)
        {
            var links = _links;
            var parts = links.GetLink(element);
            var start = links.Constants.SourcePart;
            for (var i = parts.Count - 1; i >= start; i--)
            {
                var part = parts[i];
                if (IsElement(part))
                {
                    yield return part;
                }
            }
        }
    }
}
```

1.99 ./csharp/Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

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

        private readonly Func<TLink, bool> _isElement;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LeveledSequenceWalker(ILinks<TLink> links, Func<TLink, bool> isElement) :
        ↪  base(links) => _isElement = isElement;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LeveledSequenceWalker(ILinks<TLink> links) : base(links) => _isElement =
        ↪  _links.IsPartialPoint;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IEnumerable<TLink> Walk(TLink sequence) => ToArray(sequence);

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

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

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

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Walkers
{
    public class RightSequenceWalker<TLink> : SequenceWalkerBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
        ↪  isElement) : base(links, stack, isElement) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links,
        ↪  stack, links.IsPartialPoint) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetNextElementAfterPop(TLink element) =>
        ↪  _links.GetTarget(element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetNextElementAfterPush(TLink element) =>
        ↪  _links.GetSource(element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override IEnumerable<TLink> WalkContents(TLink element)
        {
            var parts = _links.GetLink(element);
            for (var i = _links.Constants.SourcePart; i < parts.Count; i++)
            {
                var part = parts[i];
                if (IsElement(part))
                {
                    yield return part;
                }
            }
        }
    }
}
```

1.101 ./csharp/Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Walkers
{
    public abstract class SequenceWalkerBase<TLink> : LinksOperatorBase<TLink>,
    ↪  ISequenceWalker<TLink>
    {
        private readonly IStack<TLink> _stack;
        private readonly Func<TLink, bool> _isElement;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
        ↪  isElement) : base(links)
        {
            _stack = stack;
            _isElement = isElement;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : this(links,
        ↪  stack, links.IsPartialPoint) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IEnumerable<TLink> Walk(TLink sequence)
        {
            _stack.Clear();
            var element = sequence;
            if (IsElement(element))
            {
                yield return element;
            }
            else
            {
                while (true)
                {
                    if (IsElement(element))
                    {
                        if (_stack.IsEmpty)
                        {
                            break;
                        }
                        element = _stack.Pop();
                        foreach (var output in WalkContents(element))
                        {
                            yield return output;
                        }
                        element = GetNextElementAfterPop(element);
                    }
                    else
                    {
                        _stack.Push(element);
                        element = GetNextElementAfterPush(element);
                    }
                }
            }
        }

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

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

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

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

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

        private readonly TLink _stack;

        public bool IsEmpty
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _equalityComparer.Equals(Peek(), _stack);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Stack(ILinks<TLink> links, TLink stack) : base(links) => _stack = stack;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private TLink GetStackMarker() => _links.GetSource(_stack);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private TLink GetTop() => _links.GetTarget(_stack);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Peek() => _links.GetTarget(GetTop());

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }

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

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

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

1.104 ./csharp/Platform.Data.Doublets/SynchronizedLinks.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Data.Doublets;
using Platform.Threading.Synchronization;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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<TLinkAddress> : ISynchronizedLinks<TLinkAddress>
    {
        public LinksConstants<TLinkAddress> Constants
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        public ISynchronization SyncRoot
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        public ILinks<TLinkAddress> Sync
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        public ILinks<TLinkAddress> Unsync
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SynchronizedLinks(ILinks<TLinkAddress> links) : this(new
         ↪  ReaderWriterLockSynchronization(), links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SynchronizedLinks(ISynchronization synchronization, ILinks<TLinkAddress> links)
        {
            SyncRoot = synchronization;
            Sync = this;
            Unsync = links;
            Constants = links.Constants;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Count(IList<TLinkAddress> restriction) =>
        ↪   SyncRoot.ExecuteReadOperation(restriction, Unsync.Count);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler,
        ↪   IList<TLinkAddress> restrictions) => SyncRoot.ExecuteReadOperation(handler,
        ↪   restrictions, (handler1, restrictions1) => Unsync.Each(handler1, restrictions1));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Create(IList<TLinkAddress> restrictions) =>
        ↪   SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Create);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress>
        ↪   substitution) => SyncRoot.ExecuteWriteOperation(restrictions, substitution,
        ↪   Unsync.Update);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Delete(IList<TLinkAddress> restrictions) =>
        ↪   SyncRoot.ExecuteWriteOperation(restrictions, 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);
        //}
    }
}
```

1.105 ./csharp/Platform.Data.Doublets/UInt64LinksExtensions.cs

```
using System;
using System.Text;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Singletons;
using Platform.Data.Doublets.Unicode;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
         ↪  Func<Link<ulong>, 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();
        }

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

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

1.106 ./csharp/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;
using Platform.Exceptions;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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 &amp; TransactionIdCombined;
        ///         }
        ///     }
        ///
        ///     public UniqueTimestamp Timestamp
        ///     {
        ///         get
        ///         {
        ///             return (UniqueTimestamp)mask &amp; 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 : IEquatable<Transition>
        {
            public static readonly long Size = Structure<Transition>.Size;

            public readonly ulong TransactionId;
            public readonly Link<ulong> Before;
            public readonly Link<ulong> After;
            public readonly Timestamp Timestamp;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
             ↪  transactionId, Link<ulong> before, Link<ulong> after)
            {
                TransactionId = transactionId;
                Before = before;
                After = after;
                Timestamp = uniqueTimestampFactory.Create();
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
             ↪  transactionId, Link<ulong> before) : this(uniqueTimestampFactory, transactionId,
             ↪  before, default) { }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
             ↪  transactionId) : this(uniqueTimestampFactory, transactionId, default, default) {
             ↪  }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
             ↪  {After}";

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public override bool Equals(object obj) => obj is Transition transition ?
             ↪  Equals(transition) : false;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public override int GetHashCode() => (TransactionId, Before, After,
             ↪  Timestamp).GetHashCode();

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public bool Equals(Transition other) => TransactionId == other.TransactionId &&
            ↪  Before == other.Before && After == other.After && Timestamp == other.Timestamp;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static bool operator ==(Transition left, Transition right) =>
            ↪  left.Equals(right);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static bool operator !=(Transition left, Transition right) => !(left ==
            ↪  right);
        }

        /// <remarks>
        /// Другие варианты реализации транзакций (атомарности):
        ///     1. Разделение хранения значения связи ((Source Target) или (Source Linker
        ↪  Target)) и индексов.
        ///     2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
        ↪  потребуется решить вопрос
        ///        со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
        ↪  пересечениями идентификаторов.
        ///
        /// Где хранить промежуточный список транзакций?
        ///
        /// В оперативной памяти:
        ///  Минусы:
        ///     1. Может усложнить систему, если она будет функционировать самостоятельно,
        ///     так как нужно отдельно выделять память под список трансформаций.
        ///     2. Выделенной оперативной памяти может не хватить, в том случае,
        ///     если транзакция использует слишком много трансформаций.
        ///         -> Можно использовать жёсткий диск для слишком длинных транзакций.
        ///         -> Максимальный размер списка трансформаций можно ограничить / задать
        ↪  константой.
        ///     3. При подтверждении транзакции (Commit) все трансформации записываются разом
        ↪  создавая задержку.
        ///
        /// На жёстком диске:
        ///  Минусы:
        ///     1. Длительный отклик, на запись каждой трансформации.
        ///     2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
        ///         -> Это может решаться упаковкой/исключением дублирующих операций.
        ///         -> Также это может решаться тем, что короткие транзакции вообще
        ///            не будут записываться в случае отката.
        ///     3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
        ↪  операции (трансформации)
        ///        будут записаны в лог.
        ///
        /// </remarks>
        public class Transaction : DisposableBase
        {
            private readonly Queue<Transition> _transitions;
            private readonly UInt64LinksTransactionsLayer _layer;
            public bool IsCommitted { get; private set; }
            public bool IsReverted { get; private set; }

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
            : base(links)
        {
            if (string.IsNullOrWhiteSpace(logAddress))
            {
                throw new ArgumentNullException(nameof(logAddress));
            }
            // В первой строке файла хранится последняя закоммиченную транзакцию.
            // При запуске это используется для проверки удачного закрытия файла лога.
            // In the first line of the file the last committed transaction is stored.
            // On startup, this is used to check that the log file is successfully closed.
            var lastCommitedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
            var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
            if (!lastCommitedTransition.Equals(lastWrittenTransition))
            {
                Dispose();
                throw new NotSupportedException("Database is damaged, autorecovery is not
                 ↪  supported yet.");
            }
            if (lastCommitedTransition == default)
            {
                FileHelpers.WriteFirst(logAddress, lastCommitedTransition);
            }
            _lastCommitedTransition = lastCommitedTransition;
            // TODO: Think about a better way to calculate or store this value
            var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
            _lastCommitedTransactionId = allTransitions.Length > 0 ? allTransitions.Max(x =>
             ↪  x.TransactionId) : 0;
            _uniqueTimestampFactory = new UniqueTimestampFactory();
            _logAddress = logAddress;
            _log = FileHelpers.Append(logAddress);
            _transitions = new Queue<Transition>();
            _transitionsPusher = new Task(TransitionsPusher);
            _transitionsPusher.Start();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IList<ulong> GetLinkValue(ulong link) => _links.GetLink(link);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override ulong Create(IList<ulong> restrictions)
        {
            var createdLinkIndex = _links.Create();
            var createdLink = new Link<ulong>(_links.GetLink(createdLinkIndex));
            CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
             ↪  default, createdLink));
            return createdLinkIndex;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
        {
            var linkIndex = restrictions[_constants.IndexPart];
            var beforeLink = new Link<ulong>(_links.GetLink(linkIndex));
            linkIndex = _links.Update(restrictions, substitution);
            var afterLink = new Link<ulong>(_links.GetLink(linkIndex));
            CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
             ↪  beforeLink, afterLink));
            return linkIndex;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override void Delete(IList<ulong> restrictions)
        {
            var link = restrictions[_constants.IndexPart];
            var deletedLink = new Link<ulong>(_links.GetLink(link));
            _links.Delete(link);
            CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
             ↪  deletedLink, default));
        }

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void ResetCurrentTransation()
        {
            _currentTransactionId = 0;
            _currentTransactionTransitions = null;
            _currentTransaction = null;
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void TransitionsPusher()
        {
            while (!Disposable.IsDisposed && _transitionsPusher != null)
            {
                Thread.Sleep(DefaultPushDelay);
                PushTransitions();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Transaction BeginTransaction() => new Transaction(this);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void DisposeTransitions()
        {
            try
            {
                var pusher = _transitionsPusher;
                if (pusher != null)
                {
                    _transitionsPusher = null;
                    pusher.Wait();
                }
                if (_transitions != null)
                {
                    PushTransitions();
                }
                _log.DisposeIfPossible();
                FileHelpers.WriteFirst(_logAddress, _lastCommitedTransition);
            }
            catch (Exception ex)
            {
                ex.Ignore();
            }
        }

        #region DisposalBase

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                DisposeTransitions();
            }
            base.Dispose(manual, wasDisposed);
        }

        #endregion
    }
}
```

1.107 ./csharp/Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Unicode
{
    public class CharToUnicodeSymbolConverter<TLink> : LinksOperatorBase<TLink>,
     ↪  IConverter<char, TLink>
    {
        private static readonly UncheckedConverter<char, TLink> _charToAddressConverter =
         ↪  UncheckedConverter<char, TLink>.Default;

        private readonly IConverter<TLink> _addressToNumberConverter;
        private readonly TLink _unicodeSymbolMarker;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CharToUnicodeSymbolConverter(ILinks<TLink> links, IConverter<TLink>
         ↪  addressToNumberConverter, TLink unicodeSymbolMarker) : base(links)
        {
            _addressToNumberConverter = addressToNumberConverter;
            _unicodeSymbolMarker = unicodeSymbolMarker;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(char source)
        {
            var unaryNumber =
             ↪  _addressToNumberConverter.Convert(_charToAddressConverter.Convert(source));
            return _links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
        }
    }
}
```

1.108 ./csharp/Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Converters;
using Platform.Data.Doublets.Sequences.Indexes;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Unicode
{
    public class StringToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
     ↪  IConverter<string, TLink>
    {
        private readonly IConverter<char, TLink> _charToUnicodeSymbolConverter;
        private readonly ISequenceIndex<TLink> _index;
        private readonly IConverter<IList<TLink>, TLink> _listToSequenceLinkConverter;
        private readonly TLink _unicodeSequenceMarker;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
         ↪  charToUnicodeSymbolConverter, ISequenceIndex<TLink> index, IConverter<IList<TLink>,
         ↪  TLink> listToSequenceLinkConverter, TLink unicodeSequenceMarker) : base(links)
        {
            _charToUnicodeSymbolConverter = charToUnicodeSymbolConverter;
            _index = index;
            _listToSequenceLinkConverter = listToSequenceLinkConverter;
            _unicodeSequenceMarker = unicodeSequenceMarker;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(string source)
        {
            var elements = new TLink[source.Length];
            for (int i = 0; i < elements.Length; i++)
            {
                elements[i] = _charToUnicodeSymbolConverter.Convert(source[i]);
            }
            _index.Add(elements);
            var sequence = _listToSequenceLinkConverter.Convert(elements);
            return _links.GetOrCreate(sequence, _unicodeSequenceMarker);
        }
    }
}
```

1.109 ./csharp/Platform.Data.Doublets/Unicode/UnicodeMap.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnicodeMap(ILinks<ulong> links) => _links = links;

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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();
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }
    }
}
```

1.110 ./csharp/Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs

```
using System;
using System.Linq;
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Converters;
using Platform.Data.Doublets.Sequences.Walkers;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Unicode
{
    public class UnicodeSequenceToStringConverter<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<TLink, string>
    {
        private readonly ICriterionMatcher<TLink> _unicodeSequenceCriterionMatcher;
        private readonly ISequenceWalker<TLink> _sequenceWalker;
        private readonly IConverter<TLink, char> _unicodeSymbolToCharConverter;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnicodeSequenceToStringConverter(ILinks<TLink> links, ICriterionMatcher<TLink>
         ↪  unicodeSequenceCriterionMatcher, ISequenceWalker<TLink> sequenceWalker,
         ↪  IConverter<TLink, char> unicodeSymbolToCharConverter) : base(links)
        {
            _unicodeSequenceCriterionMatcher = unicodeSequenceCriterionMatcher;
            _sequenceWalker = sequenceWalker;
            _unicodeSymbolToCharConverter = unicodeSymbolToCharConverter;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public string Convert(TLink source)
        {
            if (!_unicodeSequenceCriterionMatcher.IsMatched(source))
            {
                throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
                 ↪  not a unicode sequence.");
            }
            var sequence = _links.GetSource(source);
            var charArray = _sequenceWalker.Walk(sequence).Select(_unicodeSymbolToCharConverter.
             ↪  Convert).ToArray();
            return new string(charArray);
        }
    }
}
```

1.111 ./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs

```
using System;
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Converters;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Unicode
{
    public class UnicodeSymbolToCharConverter<TLink> : LinksOperatorBase<TLink>,
     ↪  IConverter<TLink, char>
    {
        private static readonly UncheckedConverter<TLink, char> _addressToCharConverter =
         ↪  UncheckedConverter<TLink, char>.Default;

        private readonly IConverter<TLink> _numberToAddressConverter;
        private readonly ICriterionMatcher<TLink> _unicodeSymbolCriterionMatcher;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnicodeSymbolToCharConverter(ILinks<TLink> links, IConverter<TLink>
         ↪  numberToAddressConverter, ICriterionMatcher<TLink> unicodeSymbolCriterionMatcher) :
         ↪  base(links)
        {
            _numberToAddressConverter = numberToAddressConverter;
            _unicodeSymbolCriterionMatcher = unicodeSymbolCriterionMatcher;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public char Convert(TLink source)
        {
            if (!_unicodeSymbolCriterionMatcher.IsMatched(source))
            {
                throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
                 ↪  not a unicode symbol.");
            }
            return _addressToCharConverter.Convert(_numberToAddressConverter.Convert(_links.GetS
             ↪  ource(source)));
        }
    }
}
```

1.112 ./csharp/Platform.Data.Doublets.Tests/GenericLinksTests.cs

```
using System;
using Xunit;
using Platform.Reflection;
using Platform.Memory;
using Platform.Scopes;
using Platform.Data.Doublets.Memory.United.Generic;

namespace Platform.Data.Doublets.Tests
{
    public unsafe static class GenericLinksTests
    {
        [Fact]
        public static void CRUDTest()
        {
            Using<byte>(links => links.TestCRUDOperations());
            Using<ushort>(links => links.TestCRUDOperations());
            Using<uint>(links => links.TestCRUDOperations());
            Using<ulong>(links => links.TestCRUDOperations());
        }

        [Fact]
        public static void RawNumbersCRUDTest()
        {
            Using<byte>(links => links.TestRawNumbersCRUDOperations());
            Using<ushort>(links => links.TestRawNumbersCRUDOperations());
            Using<uint>(links => links.TestRawNumbersCRUDOperations());
            Using<ulong>(links => links.TestRawNumbersCRUDOperations());
        }

        [Fact]
        public static void MultipleRandomCreationsAndDeletionsTest()
        {
            Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
            ↪  MultipleRandomCreationsAndDeletions(16)); // Cannot use more because current
            ↪  implementation of tree cuts out 5 bits from the address space.
            Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Te
            ↪  stMultipleRandomCreationsAndDeletions(100));
            Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
            ↪  MultipleRandomCreationsAndDeletions(100));
            Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
            ↪  tMultipleRandomCreationsAndDeletions(100));
        }

        private static void Using<TLink>(Action<ILinks<TLink>> action)
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪  UnitedMemoryLinks<TLink>>>())
            {
                action(scope.Use<ILinks<TLink>>());
            }
        }
    }
}
```

1.113 ./csharp/Platform.Data.Doublets.Tests/ILinksExtensionsTests.cs

```
using Xunit;

namespace Platform.Data.Doublets.Tests
{
    public class ILinksExtensionsTests
    {
        [Fact]
        public void FormatTest()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                var link = links.Create();
                var linkString = links.Format(link);
                Assert.Equal("(1: 1 1)", linkString);
            }
        }
    }
}
```

1.114 ./csharp/Platform.Data.Doublets.Tests/LinksConstantsTests.cs

```
using Xunit;

namespace Platform.Data.Doublets.Tests
{
    public static class LinksConstantsTests
    {
        [Fact]
        public static void ExternalReferencesTest()
        {
            LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
            ↪  (long.MaxValue + 1UL, ulong.MaxValue));

            //var minimum = new Hybrid<ulong>(0, isExternal: true);
            var minimum = new Hybrid<ulong>(1, isExternal: true);
            var maximum = new Hybrid<ulong>(long.MaxValue, isExternal: true);

            Assert.True(constants.IsExternalReference(minimum));
            Assert.True(constants.IsExternalReference(maximum));
        }
    }
}
```

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

```
using System;
using System.Linq;
using Xunit;
using Platform.Collections.Stacks;
using Platform.Collections.Arrays;
using Platform.Memory;
using Platform.Data.Numbers.Raw;
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.PropertyOperators;
using Platform.Data.Doublets.Incrementers;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Sequences.Indexes;
using Platform.Data.Doublets.Unicode;
using Platform.Data.Doublets.Numbers.Unary;
using Platform.Data.Doublets.Decorators;
using Platform.Data.Doublets.Memory.United.Specific;

namespace Platform.Data.Doublets.Tests
{
    public static class OptimalVariantSequenceTests
    {
        private static readonly string _sequenceExample = "зеленела зелёная зелень";
        private static readonly string _loremIpsumExample = @"Lorem ipsum dolor sit amet,
         ↪  consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore
         ↪  magna aliqua.
Facilisi nullam vehicula ipsum a arcu cursus vitae congue mauris.
Et malesuada fames ac turpis egestas sed.
Eget velit aliquet sagittis id consectetur purus.
Dignissim cras tincidunt lobortis feugiat vivamus.
Vitae aliquet nec ullamcorper sit.
Lectus quam id leo in vitae.
Tortor dignissim convallis aenean et tortor at risus viverra adipiscing.
Sed risus ultricies tristique nulla aliquet enim tortor at auctor.
Integer eget aliquet nibh praesent tristique.
Vitae congue eu consequat ac felis donec et odio.
Tristique et egestas quis ipsum suspendisse.
Suspendisse potenti nullam ac tortor vitae purus faucibus ornare.
Nulla facilisi etiam dignissim diam quis enim lobortis scelerisque.
Imperdiet proin fermentum leo vel orci.
In ante metus dictum at tempor commodo.
Nisi lacus sed viverra tellus in.
Quam vulputate dignissim suspendisse in.
Elit scelerisque mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus.
Gravida cum sociis natoque penatibus et magnis dis parturient.
Risus quis varius quam quisque id diam.
Congue nisi vitae suscipit tellus mauris a diam maecenas.
Eget nunc scelerisque viverra mauris in aliquam sem fringilla.
Pharetra vel turpis nunc eget lorem dolor sed viverra.
Mattis pellentesque id nibh tortor id aliquet.
Purus non enim praesent elementum facilisis leo vel.
Etiam sit amet nisl purus in mollis nunc sed.
Tortor at auctor urna nunc id cursus metus aliquam.
Volutpat odio facilisis mauris sit amet.
Turpis egestas pretium aenean pharetra magna ac placerat.
Fermentum dui faucibus in ornare quam viverra orci sagittis eu.
Porttitor leo a diam sollicitudin tempor id eu.
Volutpat sed cras ornare arcu dui.
Ut aliquam purus sit amet luctus venenatis lectus magna.
Aliquet risus feugiat in ante metus dictum at.
Mattis nunc sed blandit libero.
Elit pellentesque habitant morbi tristique senectus et netus.
Nibh sit amet commodo nulla facilisi nullam vehicula ipsum a.
Enim sit amet venenatis urna cursus eget nunc scelerisque viverra.
Amet venenatis urna cursus eget nunc scelerisque viverra mauris in.
Diam donec adipiscing tristique risus nec feugiat.
Pulvinar mattis nunc sed blandit libero volutpat.
Cras fermentum odio eu feugiat pretium nibh ipsum.
In nulla posuere sollicitudin aliquam ultrices sagittis orci a.
Mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus et.
A iaculis at erat pellentesque.
Morbi blandit cursus risus at ultrices mi tempus imperdiet nulla.
```

```csharp
Eget lorem dolor sed viverra ipsum nunc.
Leo a diam sollicitudin tempor id eu.
Interdum consectetur libero id faucibus nisl tincidunt eget nullam non.";

        [Fact]
        public static void LinksBasedFrequencyStoredOptimalVariantSequenceTest()
        {
            using (var scope = new TempLinksTestScope(useSequences: false))
            {
                var links = scope.Links;
                var constants = links.Constants;

                links.UseUnicode();

                var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);

                var meaningRoot = links.CreatePoint();
                var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
                var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
                var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);

                var unaryNumberToAddressConverter = new UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
                var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
                var frequencyIncrementer = new FrequencyIncrementer<ulong>(links, frequencyMarker, unaryOne, unaryNumberIncrementer);
                var frequencyPropertyOperator = new PropertyOperator<ulong>(links, frequencyPropertyMarker, frequencyMarker);
                var index = new FrequencyIncrementingSequenceIndex<ulong>(links, frequencyPropertyOperator, frequencyIncrementer);
                var linkToItsFrequencyNumberConverter = new LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator, unaryNumberToAddressConverter);
                var sequenceToItsLocalElementLevelsConverter = new SequenceToItsLocalElementLevelsConverter<ulong>(links, linkToItsFrequencyNumberConverter);
                var optimalVariantConverter = new OptimalVariantConverter<ulong>(links, sequenceToItsLocalElementLevelsConverter);

                var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() { Walker = new LeveledSequenceWalker<ulong>(links) });

                ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter, index, optimalVariantConverter);
            }
        }

        [Fact]
        public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
        {
            using (var scope = new TempLinksTestScope(useSequences: false))
            {
                var links = scope.Links;

                links.UseUnicode();

                var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);

                var totalSequenceSymbolFrequencyCounter = new TotalSequenceSymbolFrequencyCounter<ulong>(links);

                var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links, totalSequenceSymbolFrequencyCounter);

                var index = new CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
                var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache);

                var sequenceToItsLocalElementLevelsConverter = new SequenceToItsLocalElementLevelsConverter<ulong>(links, linkToItsFrequencyNumberConverter);
                var optimalVariantConverter = new OptimalVariantConverter<ulong>(links, sequenceToItsLocalElementLevelsConverter);

                var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() { Walker = new LeveledSequenceWalker<ulong>(links) });
                ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter, index, optimalVariantConverter);
            }
        }

        private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence, SequenceToItsLocalElementLevelsConverter<ulong> sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index, OptimalVariantConverter<ulong> optimalVariantConverter)
        {
            index.Add(sequence);

            var optimalVariant = optimalVariantConverter.Convert(sequence);

            var readSequence1 = sequences.ToList(optimalVariant);

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

        [Fact]
        public static void SavedSequencesOptimizationTest()
        {
            LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue), (long.MaxValue + 1UL, ulong.MaxValue));

            using (var memory = new HeapResizableDirectMemory())
            using (var disposableLinks = new UInt64UnitedMemoryLinks(memory, UInt64UnitedMemoryLinks.DefaultLinksSizeStep, constants, useAvlBasedIndex: false))
            {
                var links = new UInt64Links(disposableLinks);

                var root = links.CreatePoint();

                //var numberToAddressConverter = new RawNumberToAddressConverter<ulong>();
                var addressToNumberConverter = new AddressToRawNumberConverter<ulong>();

                var unicodeSymbolMarker = links.GetOrCreate(root, addressToNumberConverter.Convert(1));
                var unicodeSequenceMarker = links.GetOrCreate(root, addressToNumberConverter.Convert(2));

                var totalSequenceSymbolFrequencyCounter = new TotalSequenceSymbolFrequencyCounter<ulong>(links);
                var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links, totalSequenceSymbolFrequencyCounter);
                var index = new CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
                var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache);
                var sequenceToItsLocalElementLevelsConverter = new SequenceToItsLocalElementLevelsConverter<ulong>(links, linkToItsFrequencyNumberConverter);
                var optimalVariantConverter = new OptimalVariantConverter<ulong>(links, sequenceToItsLocalElementLevelsConverter);

                var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>(), (link) => constants.IsExternalReference(link) || links.IsPartialPoint(link));

                var unicodeSequencesOptions = new SequencesOptions<ulong>()
                {
                    UseSequenceMarker = true,
                    SequenceMarkerLink = unicodeSequenceMarker,
                    UseIndex = true,
                    Index = index,
                    LinksToSequenceConverter = optimalVariantConverter,
                    Walker = walker,
                    UseGarbageCollection = true
                };

                var unicodeSequences = new Sequences.Sequences(new SynchronizedLinks<ulong>(links), unicodeSequencesOptions);

                // Create some sequences
                var strings = _loremIpsumExample.Split(new[] { '\n', '\r' }, StringSplitOptions.RemoveEmptyEntries);
                var arrays = strings.Select(x => x.Select(y => addressToNumberConverter.Convert(y)).ToArray()).ToArray();
                for (int i = 0; i < arrays.Length; i++)
                {
                    unicodeSequences.Create(arrays[i].ShiftRight());
                }

                var linksCountAfterCreation = links.Count();

                // get list of sequences links
                // for each sequence link
                //   create new sequence version
                //   if new sequence is not the same as sequence link
                //     delete sequence link
                //     collect garbadge
                unicodeSequences.CompactAll();

                var linksCountAfterCompactification = links.Count();

                Assert.True(linksCountAfterCompactification < linksCountAfterCreation);
            }
        }
    }
}
```

1.116 ./csharp/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;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Sequences;

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

            using (var scope = new TempLinksTestScope(useSequences: false))
            {
                var links = scope.Links;
                var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong> {
                ↪  Walker = new LeveledSequenceWalker<ulong>(links) });

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

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

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

                var sw2 = Stopwatch.StartNew();
                var readSequence1 = sequences.ToList(balancedVariant); sw2.Stop();

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

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

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

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

                Console.WriteLine($"Stack-based walker: {sw3.Elapsed}, Level-based reader:
                ↪  {sw2.Elapsed}");
                for (var i = 0; i < sequenceLength; i++)
                {
                    links.Delete(sequence[i]);
                }
            }
        }
    }
}
```

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

```
using System.IO;
using Xunit;
using Platform.Singletons;
using Platform.Memory;
using Platform.Data.Doublets.Memory.United.Specific;

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

        [Fact]
        public static void BasicFileMappedMemoryTest()
        {
            var tempFilename = Path.GetTempFileName();
            using (var memoryAdapter = new UInt64UnitedMemoryLinks(tempFilename))
            {
                memoryAdapter.TestBasicMemoryOperations();
            }
            File.Delete(tempFilename);
        }

        [Fact]
        public static void BasicHeapMemoryTest()
        {
            using (var memory = new
             ↪  HeapResizableDirectMemory(UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
            using (var memoryAdapter = new UInt64UnitedMemoryLinks(memory,
             ↪  UInt64UnitedMemoryLinks.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(UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
            using (var memoryAdapter = new UInt64UnitedMemoryLinks(memory,
             ↪  UInt64UnitedMemoryLinks.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);
        }
    }
}
```

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

```
using Xunit;
using Platform.Scopes;
using Platform.Memory;
using Platform.Data.Doublets.Decorators;
using Platform.Reflection;
using Platform.Data.Doublets.Memory.United.Generic;
using Platform.Data.Doublets.Memory.United.Specific;

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<UInt64UnitedMemoryLinks>();
                var instance = scope.Use<ILinks<ulong>>();
                Assert.IsType<UInt64UnitedMemoryLinks>(instance);
            }
        }

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

        [Fact]
        public static void TypeParametersTest()
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
             ↪  UnitedMemoryLinks<ulong>>>())
            {
                var links = scope.Use<ILinks<ulong>>();
                Assert.IsType<UnitedMemoryLinks<ulong>>(links);
            }
        }
    }
}
```

1.119 ./csharp/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.Collections.Arrays;
using Platform.Random;
using Platform.IO;
using Platform.Singletons;
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Unicode;

namespace Platform.Data.Doublets.Tests
```

```
{
    public static class SequencesTests
    {
        private static readonly LinksConstants<ulong> _constants =
         ↪  Default<LinksConstants<ulong>>.Instance;

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

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

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

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

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

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

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

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

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

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

        //    const long sequenceLength = 8;

        //    const ulong itself = LinksConstants.Itself;

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

        //        SequencesOptions o = new SequencesOptions();

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

        //        var sequences = new Sequences(links);

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

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

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

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

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

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

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

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

                var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();

                //for (int i = 0; i < createResults.Length; i++)
                //    sequences.Create(createResults[i]);

                var sw0 = Stopwatch.StartNew();
                var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();

                var sw1 = Stopwatch.StartNew();
                var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();

                var sw2 = Stopwatch.StartNew();
                var searchResults2 = sequences.Each1(sequence); sw2.Stop();

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

                var intersection0 = createResults.Intersect(searchResults0).ToList();
                Assert.True(intersection0.Count == searchResults0.Count);
                Assert.True(intersection0.Count == createResults.Length);

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

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

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

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

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

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

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

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

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

                var sw2 = Stopwatch.StartNew();
                var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();

                var sw3 = Stopwatch.StartNew();
                var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();

                // На количестве в 200 элементов это будет занимать вечность
                //var sw4 = Stopwatch.StartNew();
                //var searchResults4 = sequences.Each(sequence); sw4.Stop();

                Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);

                Assert.True(searchResults3.Count == 1 && balancedVariant ==
                ↪  searchResults3.First());

                //Assert.True(sw1.Elapsed < sw2.Elapsed);

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

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

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

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

                var createResults = sequences.CreateAllVariants2(sequence);

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

                var partialSequence = new ulong[sequenceLength - 2];

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

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

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

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

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

                //Global.Trash = searchResults3;

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

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

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

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

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

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

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

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

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

                var balancedVariant = balancedVariantConverter.Convert(sequence);

                var partialSequence = new ulong[sequenceLength - 2];

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

                var sw1 = Stopwatch.StartNew();
                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);

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

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

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

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

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

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

                index.Add(sequence);

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

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

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

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

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

[![чёрное пространство, чёрная точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png ""чёрное пространство, чёрная точка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png)
```

```

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

[![белая точка, чёрная точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png ""белая точка, чёрная точка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png)

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

[![две белые точки, чёрная вертикальная линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png ""две белые точки, чёрная вертикальная линия"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png)

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

...

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

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

        [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()
        {
            var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
             ↪  StringSplitOptions.RemoveEmptyEntries);
            var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
            var totalCharacters = arrays.Select(x => x.Length).Sum();

            using (var scope1 = new TempLinksTestScope(useSequences: true))
            using (var scope2 = new TempLinksTestScope(useSequences: true))
            using (var scope3 = new TempLinksTestScope(useSequences: true))
            {
                scope1.Links.Unsync.UseUnicode();
                scope2.Links.Unsync.UseUnicode();
                scope3.Links.Unsync.UseUnicode();

                var balancedVariantConverter1 = new
                 ↪  BalancedVariantConverter<ulong>(scope1.Links.Unsync);
                var totalSequenceSymbolFrequencyCounter = new
                 ↪  TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
                var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
                 ↪  totalSequenceSymbolFrequencyCounter);
                var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
                 ↪  balancedVariantConverter1, linkFrequenciesCache1,
                 ↪  doInitialFrequenciesIncrement: false);

                //var compressor2 = scope2.Sequences;
                var compressor3 = scope3.Sequences;

                var constants = Default<LinksConstants<ulong>>.Instance;

                var sequences = compressor3;
                //var meaningRoot = links.CreatePoint();
                //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
                //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
                //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
                 ↪  constants.Itself);

                //var unaryNumberToAddressConverter = new
                 ↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
                //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
                 ↪  unaryOne);
                //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                 ↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
                //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
                 ↪  frequencyPropertyMarker, frequencyMarker);
                //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
                 ↪  frequencyPropertyOperator, frequencyIncrementer);
                //var linkToItsFrequencyNumberConverter = new
                 ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                 ↪  unaryNumberToAddressConverter);

                var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
                 ↪  totalSequenceSymbolFrequencyCounter);

                var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque⌟
                 ↪  ncyNumberConverter<ulong>(linkFrequenciesCache3);

                var sequenceToItsLocalElementLevelsConverter = new
                 ↪  SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
                 ↪  linkToItsFrequencyNumberConverter);
                var optimalVariantConverter = new
                 ↪  OptimalVariantConverter<ulong>(scope3.Links.Unsync,
                 ↪  sequenceToItsLocalElementLevelsConverter);

                var compressed1 = new ulong[arrays.Length];
            var compressed2 = new ulong[arrays.Length];
            var compressed3 = new ulong[arrays.Length];

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

            //for (int i = START; i < END; i++)
            //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);

            var initialCount1 = scope2.Links.Unsync.Count();

            var sw1 = Stopwatch.StartNew();

            for (int i = START; i < END; i++)
            {
                linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
                compressed1[i] = compressor1.Convert(arrays[i]);
            }

            var elapsed1 = sw1.Elapsed;

            var balancedVariantConverter2 = new
            ↪  BalancedVariantConverter<ulong>(scope2.Links.Unsync);

            var initialCount2 = scope2.Links.Unsync.Count();

            var sw2 = Stopwatch.StartNew();

            for (int i = START; i < END; i++)
            {
                compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
            }

            var elapsed2 = sw2.Elapsed;

            for (int i = START; i < END; i++)
            {
                linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
            }

            var initialCount3 = scope3.Links.Unsync.Count();

            var sw3 = Stopwatch.StartNew();

            for (int i = START; i < END; i++)
            {
                //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
                compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
            }

            var elapsed3 = sw3.Elapsed;

            Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
            ↪  Optimal variant: {elapsed3}");

            // Assert.True(elapsed1 > elapsed2);

            // Checks
            for (int i = START; i < END; i++)
            {
                var sequence1 = compressed1[i];
                var sequence2 = compressed2[i];
                var sequence3 = compressed3[i];

                var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                ↪  scope1.Links.Unsync);

                var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                ↪  scope2.Links.Unsync);

                var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
                ↪  scope3.Links.Unsync);

                var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
                ↪  link.IsPartialPoint());
                var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
                ↪  link.IsPartialPoint());
                var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
                ↪  link.IsPartialPoint());
```

```csharp

                    //if (sequence1 != Constants.Null && sequence2 != Constants.Null && arrays[i].Length > 3)
                    //    Assert.False(structure1 == structure2);
                    //if (sequence3 != Constants.Null && sequence2 != Constants.Null && arrays[i].Length > 3)
                    //    Assert.False(structure3 == structure2);

                    Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
                    Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
                }

                Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) < totalCharacters);
                Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) < totalCharacters);
                Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) < totalCharacters);

                Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) / totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) / totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) / totalCharacters}");

                Assert.True(scope1.Links.Unsync.Count() - initialCount1 < scope2.Links.Unsync.Count() - initialCount2);
                Assert.True(scope3.Links.Unsync.Count() - initialCount3 < scope2.Links.Unsync.Count() - initialCount2);

                var duplicateProvider1 = new DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
                var duplicateProvider2 = new DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
                var duplicateProvider3 = new DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);

                var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
                var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
                var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);

                var duplicates1 = duplicateCounter1.Count();

                ConsoleHelpers.Debug("------");

                var duplicates2 = duplicateCounter2.Count();

                ConsoleHelpers.Debug("------");

                var duplicates3 = duplicateCounter3.Count();

                Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");

                linkFrequenciesCache1.ValidateFrequencies();
                linkFrequenciesCache3.ValidateFrequencies();
            }
        }

        [Fact]
        public static void CompressionStabilityTest()
        {
            // TODO: Fix bug (do a separate test)
            //const ulong minNumbers = 0;
            //const ulong maxNumbers = 1000;

            const ulong minNumbers = 10000;
            const ulong maxNumbers = 12500;

            var strings = new List<string>();

            for (ulong i = minNumbers; i < maxNumbers; i++)
            {
                strings.Add(i.ToString());
            }

            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 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
                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;

                // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
                // Stability issue starts at 10001 or 11000
                //for (int i = START; i < END; i++)
                //{
                //    var first = compressor1.Compress(arrays[i]);
                //    var second = compressor1.Compress(arrays[i]);

                //    if (first == second)
                //        compressed1[i] = first;
                //    else
                //    {
                //        // TODO: Find a solution for this case
                //    }
                //}

                for (int i = START; i < END; i++)
                {
                    var first = compressor1.Create(arrays[i].ShiftRight());
                    var second = compressor1.Create(arrays[i].ShiftRight());

                    if (first == second)
                    {
                        compressed1[i] = first;
                    }
                    else
                    {
                        // TODO: Find a solution for this case
                    }
                }

                var elapsed1 = sw1.Elapsed;

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

                var sw2 = Stopwatch.StartNew();

                for (int i = START; i < END; i++)
                {
                    var first = balancedVariantConverter.Convert(arrays[i]);
                    var second = balancedVariantConverter.Convert(arrays[i]);

                    if (first == second)
                    {
                        compressed2[i] = first;
                    }
                }

                var elapsed2 = sw2.Elapsed;

                Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
                 ↪  {elapsed2}");

                Assert.True(elapsed1 > elapsed2);

                // Checks
                for (int i = START; i < END; i++)
                {
                    var sequence1 = compressed1[i];
                    var sequence2 = compressed2[i];
                    if (sequence1 != _constants.Null && sequence2 != _constants.Null)
                    {
                        var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                        ↪  scope1.Links);

                        var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                        ↪  scope2.Links);

                        //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
                        ↪  link.IsPartialPoint());
                        //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
                        ↪  link.IsPartialPoint());

                        //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].ShiftRight());
                }

                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]);
                }

                var elapsed2 = sw2.Elapsed;

                Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator: {elapsed2}");

                Assert.True(elapsed1 > elapsed2);

                // Checks
                for (int i = START; i < END; i++)
                {
                    var sequence1 = compressed1[i];
                    var sequence2 = compressed2[i];

                    if (sequence1 != _constants.Null && sequence2 != _constants.Null)
                    {
                        var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1, scope1.Links);

                        var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2, scope2.Links);

                        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}");

                // Can be worse than balanced variant
                //Assert.True(scope1.Links.Count() <= scope2.Links.Count());

                //compressor1.ValidateFrequencies();
            }
        }

        [Fact]
        public static void AllTreeBreakDownAtSequencesCreationBugTest()
        {
            // Made out of AllPossibleConnectionsTest test.

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

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

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

                var createResults = sequences.CreateAllVariants2(sequence);

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

                var createResults = sequences.CreateAllVariants2(sequence);

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

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

                    sequences.CalculateAllUsages(linksTotalUsages1);

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

                    sequences.CalculateAllUsages2(linksTotalUsages2);

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

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

1.120 ./csharp/Platform.Data.Doublets.Tests/SplitMemoryGenericLinksTests.cs

```
using System;
using Xunit;
using Platform.Memory;
using Platform.Data.Doublets.Memory.Split.Generic;

namespace Platform.Data.Doublets.Tests
{
    public unsafe static class SplitMemoryGenericLinksTests
    {
        [Fact]
        public static void CRUDTest()
        {
            Using<byte>(links => links.TestCRUDOperations());
            Using<ushort>(links => links.TestCRUDOperations());
            Using<uint>(links => links.TestCRUDOperations());
            Using<ulong>(links => links.TestCRUDOperations());
        }

        [Fact]
        public static void RawNumbersCRUDTest()
        {
            UsingWithExternalReferences<byte>(links => links.TestRawNumbersCRUDOperations());
            UsingWithExternalReferences<ushort>(links => links.TestRawNumbersCRUDOperations());
            UsingWithExternalReferences<uint>(links => links.TestRawNumbersCRUDOperations());
            UsingWithExternalReferences<ulong>(links => links.TestRawNumbersCRUDOperations());
        }

        [Fact]
        public static void MultipleRandomCreationsAndDeletionsTest()
        {
            Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test⌋
             ↪  MultipleRandomCreationsAndDeletions(16)); // Cannot use more because current
             ↪  implementation of tree cuts out 5 bits from the address space.
            Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Te⌋
             ↪  stMultipleRandomCreationsAndDeletions(100));
            Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test⌋
             ↪  MultipleRandomCreationsAndDeletions(100));
            Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes⌋
             ↪  tMultipleRandomCreationsAndDeletions(100));
        }

        private static void Using<TLink>(Action<ILinks<TLink>> action)
        {
            using (var dataMemory = new HeapResizableDirectMemory())
            using (var indexMemory = new HeapResizableDirectMemory())
            using (var memory = new SplitMemoryLinks<TLink>(dataMemory, indexMemory))
            {
                action(memory);
            }
        }

        private static void UsingWithExternalReferences<TLink>(Action<ILinks<TLink>> action)
        {
            var contants = new LinksConstants<TLink>(enableExternalReferencesSupport: true);
            using (var dataMemory = new HeapResizableDirectMemory())
            using (var indexMemory = new HeapResizableDirectMemory())
            using (var memory = new SplitMemoryLinks<TLink>(dataMemory, indexMemory,
             ↪  SplitMemoryLinks<TLink>.DefaultLinksSizeStep, contants))
            {
                action(memory);
            }
        }
    }
}
```

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

```
using System.IO;
using Platform.Disposables;
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Decorators;
using Platform.Data.Doublets.Memory.United.Specific;

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

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

        public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles =
         ↪  true, bool useSequences = false, bool useLog = false)
        {
            _deleteFiles = deleteFiles;
            TempFilename = Path.GetTempFileName();
            TempTransactionLogFilename = Path.GetTempFileName();
            var coreMemoryAdapter = new UInt64UnitedMemoryLinks(TempFilename);
            MemoryAdapter = useLog ? (ILinks<ulong>)new
             ↪  UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
             ↪  coreMemoryAdapter;
            Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
            if (useSequences)
            {
                Sequences = new Sequences.Sequences(Links, sequencesOptions);
            }
        }

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

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

1.122 ./csharp/Platform.Data.Doublets.Tests/TestExtensions.cs

```csharp
using System.Collections.Generic;
using Xunit;
using Platform.Ranges;
using Platform.Numbers;
using Platform.Random;
using Platform.Setters;
using Platform.Converters;

namespace Platform.Data.Doublets.Tests
{
    public static class TestExtensions
    {
        public static void TestCRUDOperations<T>(this ILinks<T> links)
        {
            var constants = links.Constants;

            var equalityComparer = EqualityComparer<T>.Default;

            var zero = default(T);
            var one = Arithmetic.Increment(zero);

            // Create Link
            Assert.True(equalityComparer.Equals(links.Count(), zero));

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

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

            var linkAddress = links.Create();

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

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

            Assert.True(equalityComparer.Equals(links.Count(), one));

            // Get first link
            setter = new Setter<T>(constants.Null);
            links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);

            Assert.True(equalityComparer.Equals(setter.Result, linkAddress));

            // Update link to reference itself
            links.Update(linkAddress, linkAddress, linkAddress);

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

            Assert.True(equalityComparer.Equals(link.Source, linkAddress));
            Assert.True(equalityComparer.Equals(link.Target, linkAddress));

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

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

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

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

            // Delete link
            links.Delete(linkAddress);

            Assert.True(equalityComparer.Equals(links.Count(), zero));

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

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

        public static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
        {
            // Constants
            var constants = links.Constants;
            var equalityComparer = EqualityComparer<T>.Default;
            var zero = default(T);
            var one = Arithmetic.Increment(zero);
            var two = Arithmetic.Increment(one);

            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(), two));

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

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

        public static void TestMultipleRandomCreationsAndDeletions<TLink>(this ILinks<TLink>
         ↪  links, int maximumOperationsPerCycle)
        {
            var comparer = Comparer<TLink>.Default;
            var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
            var uInt64ToAddressConverter = CheckedConverter<ulong, TLink>.Default;
            for (var N = 1; N < maximumOperationsPerCycle; N++)
            {
                var random = new System.Random(N);
                var created = 0UL;
                var deleted = 0UL;
                for (var i = 0; i < N; i++)
                {
                    var linksCount = addressToUInt64Converter.Convert(links.Count());
                    var createPoint = random.NextBoolean();
                    if (linksCount > 2 && createPoint)
                    {
                        var linksAddressRange = new Range<ulong>(1, linksCount);
                        TLink source = uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
                        TLink target = uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange)); //-V3086
                        var resultLink = links.GetOrCreate(source, target);
                        if (comparer.Compare(resultLink, uInt64ToAddressConverter.Convert(linksCount)) > 0)
                        {
                            created++;
                        }
                    }
                    else
                    {
                        links.Create();
                        created++;
                    }
                }
                Assert.True(created == addressToUInt64Converter.Convert(links.Count()));
                for (var i = 0; i < N; i++)
                {
                    TLink link = uInt64ToAddressConverter.Convert((ulong)i + 1UL);
                    if (links.Exists(link))
                    {
                        links.Delete(link);
                        deleted++;
                    }
                }
                Assert.True(addressToUInt64Converter.Convert(links.Count()) == 0L);
            }
        }
    }
}
```

1.123 ./csharp/Platform.Data.Doublets.Tests/UInt64LinksTests.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.Ranges;
using Platform.Random;
using Platform.Timestamps;
using Platform.Reflection;
using Platform.Singletons;
using Platform.Scopes;
using Platform.Counters;
using Platform.Diagnostics;
using Platform.IO;
using Platform.Memory;
using Platform.Data.Doublets.Decorators;
using Platform.Data.Doublets.Memory.United.Specific;

namespace Platform.Data.Doublets.Tests
{
    public static class UInt64LinksTests
    {
        private static readonly LinksConstants<ulong> _constants = Default<LinksConstants<ulong>>.Instance;

        private const long Iterations = 10 * 1024;

        #region Concept

        [Fact]
        public static void MultipleCreateAndDeleteTest()
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪  UInt64UnitedMemoryLinks>>())
            {
                new UInt64Links(scope.Use<ILinks<ulong>>()).TestMultipleRandomCreationsAndDeleti⌟
                ↪  ons(100);
            }
        }

        [Fact]
        public static void CascadeUpdateTest()
        {
            var itself = _constants.Itself;
            using (var scope = new TempLinksTestScope(useLog: true))
            {
                var links = scope.Links;

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

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

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

                l2 = links.Update(l2, l1);

                links.Delete(l2);

                Global.Trash = links.Count();

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

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

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

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

                links.Delete(l1);

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

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

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

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

                Assert.Equal(0UL, links.Count());
                links.Unsync.DisposeIfPossible();

                var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope.TempTransactionLogFilename);
                Assert.Single(transitions);
            }
        }

        [Fact]
        public static void TransactionUserCodeErrorNoDataSavedTest()
        {
            // User Code Error (Autoreverted), no data saved
            var itself = _constants.Itself;

            TempLinksTestScope lastScope = null;
            try
            {
                using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false, useLog: true))
                {
                    var links = scope.Links;
                    var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecoratorBase<ulong>)links.Unsync).Links;
                    using (var transaction = transactionsLayer.BeginTransaction())
                    {
                        var l1 = links.CreateAndUpdate(itself, itself);
                        var l2 = links.CreateAndUpdate(itself, itself);

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

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

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

                        l2 = links.Update(l2, l1);

                        links.Delete(l2);

                        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>(last
                 ↪  Scope.TempTransactionLogFilename);

                lastScope.DeleteFiles();
            }
        }

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

            var tempDatabaseFilename = Path.GetTempFileName();
            var tempTransactionLogFilename = Path.GetTempFileName();

            // Commit
            using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
             ↪  UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
            using (var links = new UInt64Links(memoryAdapter))
            {
                using (var transaction = memoryAdapter.BeginTransaction())
                {
                    var l1 = links.CreateAndUpdate(itself, itself);
                    var l2 = links.CreateAndUpdate(itself, itself);

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

                    links.Delete(l1);

                    transaction.Commit();
                }

                Global.Trash = links.Count();
            }

            Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
             ↪  sactionLogFilename);
        }

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

```csharp

            var tempDatabaseFilename = Path.GetTempFileName();
            var tempTransactionLogFilename = Path.GetTempFileName();

            // Commit
            using (var memoryAdapter = new UInt64LinksTransactionsLayer(new UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
            using (var links = new UInt64Links(memoryAdapter))
            {
                using (var transaction = memoryAdapter.BeginTransaction())
                {
                    var l1 = links.CreateAndUpdate(itself, itself);
                    var l2 = links.CreateAndUpdate(itself, itself);

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

                    links.Delete(l1);

                    transaction.Commit();
                }

                Global.Trash = links.Count();
            }

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

            // Damage database

            FileHelpers.WriteFirst(tempTransactionLogFilename, new UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));

            // Try load damaged database
            try
            {
                // TODO: Fix
                using (var memoryAdapter = new UInt64LinksTransactionsLayer(new UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
                using (var links = new UInt64Links(memoryAdapter))
                {
                    Global.Trash = links.Count();
                }
            }
            catch (NotSupportedException ex)
            {
                Assert.True(ex.Message == "Database is damaged, autorecovery is not supported yet.");
            }

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

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

        [Fact]
        public static void Bug1Test()
        {
            var tempDatabaseFilename = Path.GetTempFileName();
            var tempTransactionLogFilename = Path.GetTempFileName();

            var itself = _constants.Itself;

            // User Code Error (Autoreverted), some data saved
            try
            {
                ulong l1;
                ulong l2;

                using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))
                using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory, tempTransactionLogFilename))
                using (var links = new UInt64Links(memoryAdapter))
                {
                    l1 = links.CreateAndUpdate(itself, itself);
                    l2 = links.CreateAndUpdate(itself, itself);

                    l2 = links.Update(l2, l2, l1, l2);
                    links.CreateAndUpdate(l2, itself);
                    links.CreateAndUpdate(l2, itself);
                }

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

                using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))
                using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
                ↪  tempTransactionLogFilename))
                using (var links = new UInt64Links(memoryAdapter))
                {
                    using (var transaction = memoryAdapter.BeginTransaction())
                    {
                        l2 = links.Update(l2, l1);

                        links.Delete(l2);

                        ExceptionThrower();

                        transaction.Commit();
                    }

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

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

        private static void ExceptionThrower() => throw new InvalidOperationException();

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

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

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

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

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

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

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

                Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
                Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
                Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));

                Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
                 ↪  "(5:(4:5 (6:5 4)) 6)");
                Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
                 ↪  "(6:(5:(4:5 6) 6) 4)");
                Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
                 ↪  "(4:(5:4 (6:5 4)) 6)");

                // 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)"

                Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
                 ↪  "{{5}{5}{4}{6}}");
                Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
                 ↪  "{{5}{6}{6}{4}}");
                Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
                 ↪  "{{4}{5}{4}{6}}");
            }
        }

        private static void DefaultFormatter(StringBuilder sb, ulong link)
        {
            sb.Append(link.ToString());
        }

        #endregion

        #region Performance

        /*
       public static void RunAllPerformanceTests()
       {
           try
           {
               links.TestLinksInSteps();
           }
           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)));

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

            ConsoleHelpers.Debug();

            ConsoleHelpers.Debug("C S D");

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

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

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

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

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

         private static TimeSpan GetBaseRandomLoopOverhead(long loops)
         {
             return Measure(() =>
             {
                 ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
                 ulong result = 0;
                 for (long i = 0; i < loops; i++)
                 {
                     var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
                     var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
                result += maxValue + source + target;
            }
            Global.Trash = result;
        });
    }
     */

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

            ulong counter = 0;

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

            var sw = Stopwatch.StartNew();

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

            var elapsedTime = sw.Elapsed;

            var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

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

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

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

            long counter = 0;

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

            var sw = Stopwatch.StartNew();

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

            var elapsedTime = sw.Elapsed;

            var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

            links.Delete(firstLink);

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

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

                ulong counter = 0;

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

                var sw = Stopwatch.StartNew();

                for (ulong i = 0; i < Iterations; i++)
                {
                    counter += links.GetTarget(firstLink);
                }

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

                links.Delete(firstLink);

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

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

                long counter = 0;

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

                var sw = Stopwatch.StartNew();

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

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

                links.Delete(firstLink);

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

        // TODO: Заполнить базу данных перед тестом
        /*
        [Fact]
        public void TestRandomSearchFixed()
        {
            var tempFilename = Path.GetTempFileName();

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

```
            {
                long iterations = 64 * 1024 * 1024 / Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;

                ulong counter = 0;
                var maxLink = links.Total;

                ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);

                var sw = Stopwatch.StartNew();

                for (var i = iterations; i > 0; i--)
                {
                    var source = RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
                    var target = RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);

                    counter += links.Search(source, target);
                }

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;

                ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2} Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond, counter);
            }

            File.Delete(tempFilename);
        }*/

        [Fact(Skip = "useless: O(0), was dependent on creation tests")]
        public static void TestRandomSearchAll()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                ulong counter = 0;

                var maxLink = links.Count();

                var iterations = links.Count();

                ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", links.Count());

                var sw = Stopwatch.StartNew();

                for (var i = iterations; i > 0; i--)
                {
                    var linksAddressRange = new Range<ulong>(_constants.InternalReferencesRange.Minimum, maxLink);

                    var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
                    var target = RandomHelpers.Default.NextUInt64(linksAddressRange);

                    counter += links.SearchOrDefault(source, target);
                }

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;

                ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2} Iterations per second), c: {3}",
                     iterations, elapsedTime, (long)iterationsPerSecond, counter);
            }
        }

        [Fact(Skip = "useless: O(0), was dependent on creation tests")]
        public static void TestEach()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;

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

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

                var sw = Stopwatch.StartNew();

                links.Each(counter.IncrementAndReturnTrue);

                var elapsedTime = sw.Elapsed;

                var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;

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

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

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

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

                var sw = Stopwatch.StartNew();

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

                var elapsedTime = sw.Elapsed;

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

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

            File.Delete(tempFilename);
        }
        */

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

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

                long counter = 0;

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

                var sw = Stopwatch.StartNew();

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

                var elapsedTime = sw.Elapsed;

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

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

            File.Delete(tempFilename);
        }
        */

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

                long linksToCreate = 64 * 1024 * 1024 / UInt64UnitedMemoryLinks.LinkSizeInBytes;

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

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

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

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

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

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

                var sw = Stopwatch.StartNew();

                long linksToCreate = 64 * 1024 * 1024 / UInt64UnitedMemoryLinks.LinkSizeInBytes;

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

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

                var elapsedTime = sw.Elapsed;

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

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

        [Fact(Skip = "useless: O(0), was dependent on creation tests")]
        public static void TestDeletionOfAllLinks()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                var linksBeforeTest = links.Count();

                ConsoleHelpers.Debug("Deleting all links");

                var elapsedTime = Performance.Measure(links.DeleteAll);

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

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

        #endregion
    }
}
```

1.124 ./csharp/Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs

```
using Xunit;
using Platform.Random;
using Platform.Data.Doublets.Numbers.Unary;

namespace Platform.Data.Doublets.Tests
{
    public static class UnaryNumberConvertersTests
    {
        [Fact]
        public static void ConvertersTest()
        {
            using (var scope = new TempLinksTestScope())
            {
                const int N = 10;
                var links = scope.Links;
                var meaningRoot = links.CreatePoint();
                var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
                var powerOf2ToUnaryNumberConverter = new
                ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, one);
                var toUnaryNumberConverter = new AddressToUnaryNumberConverter<ulong>(links,
                ↪  powerOf2ToUnaryNumberConverter);
                var random = new System.Random(0);
                ulong[] numbers = new ulong[N];
                ulong[] unaryNumbers = new ulong[N];
                for (int i = 0; i < N; i++)
                {
                    numbers[i] = random.NextUInt64();
                    unaryNumbers[i] = toUnaryNumberConverter.Convert(numbers[i]);
                }
                var fromUnaryNumberConverterUsingOrOperation = new
                ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links,
                ↪  powerOf2ToUnaryNumberConverter);
                var fromUnaryNumberConverterUsingAddOperation = new
                ↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, one);
                for (int i = 0; i < N; i++)
                {
                    Assert.Equal(numbers[i],
                    ↪  fromUnaryNumberConverterUsingOrOperation.Convert(unaryNumbers[i]));
                    Assert.Equal(numbers[i],
                    ↪  fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));
                }
            }
        }
    }
}
```

1.125 ./csharp/Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs

```
using Xunit;
using Platform.Converters;
using Platform.Memory;
using Platform.Reflection;
using Platform.Scopes;
using Platform.Data.Numbers.Raw;
using Platform.Data.Doublets.Incrementers;
using Platform.Data.Doublets.Numbers.Unary;
using Platform.Data.Doublets.PropertyOperators;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Sequences.Indexes;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Unicode;
using Platform.Data.Doublets.Memory.United.Generic;
using Platform.Data.Doublets.CriterionMatchers;

namespace Platform.Data.Doublets.Tests
{
    public static class UnicodeConvertersTests
    {
        [Fact]
        public static void CharAndUnaryNumberUnicodeSymbolConvertersTest()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                var meaningRoot = links.CreatePoint();
                var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
                var powerOf2ToUnaryNumberConverter = new
                ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, one);
                var addressToUnaryNumberConverter = new
                ↪  AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
                var unaryNumberToAddressConverter = new
                ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links,
                ↪  powerOf2ToUnaryNumberConverter);
                TestCharAndUnicodeSymbolConverters(links, meaningRoot,
                ↪  addressToUnaryNumberConverter, unaryNumberToAddressConverter);
            }
        }

        [Fact]
        public static void CharAndRawNumberUnicodeSymbolConvertersTest()
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪  UnitedMemoryLinks<ulong>>>())
            {
                var links = scope.Use<ILinks<ulong>>();
                var meaningRoot = links.CreatePoint();
                var addressToRawNumberConverter = new AddressToRawNumberConverter<ulong>();
                var rawNumberToAddressConverter = new RawNumberToAddressConverter<ulong>();
                TestCharAndUnicodeSymbolConverters(links, meaningRoot,
                ↪  addressToRawNumberConverter, rawNumberToAddressConverter);
            }
        }

        private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
        ↪  meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
        ↪  numberToAddressConverter)
        {
            var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
            var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links,
            ↪  addressToNumberConverter, unicodeSymbolMarker);
            var originalCharacter = 'H';
            var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);
            var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links,
            ↪  unicodeSymbolMarker);
            var unicodeSymbolToCharConverter = new UnicodeSymbolToCharConverter<ulong>(links,
            ↪  numberToAddressConverter, unicodeSymbolCriterionMatcher);
            var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
            Assert.Equal(originalCharacter, resultingCharacter);
        }

        [Fact]
        public static void StringAndUnicodeSequenceConvertersTest()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;

                var itself = links.Constants.Itself;

                var meaningRoot = links.CreatePoint();
                var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
                var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
                var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
                var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
                var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);

                var powerOf2ToUnaryNumberConverter = new
                ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
                var addressToUnaryNumberConverter = new
                ↪  AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
                var charToUnicodeSymbolConverter = new
                ↪  CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
                ↪  unicodeSymbolMarker);

                var unaryNumberToAddressConverter = new
                ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links,
                ↪  powerOf2ToUnaryNumberConverter);
                var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
                var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                ↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
                var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
                ↪  frequencyPropertyMarker, frequencyMarker);
                var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
                ↪  frequencyPropertyOperator, frequencyIncrementer);
                var linkToItsFrequencyNumberConverter = new
                ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                ↪  unaryNumberToAddressConverter);
                var sequenceToItsLocalElementLevelsConverter = new
                ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                ↪  linkToItsFrequencyNumberConverter);
                var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                ↪  sequenceToItsLocalElementLevelsConverter);

                var stringToUnicodeSequenceConverter = new
                ↪  StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
                ↪  index, optimalVariantConverter, unicodeSequenceMarker);

                var originalString = "Hello";

                var unicodeSequenceLink =
                ↪  stringToUnicodeSequenceConverter.Convert(originalString);

                var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links,
                ↪  unicodeSymbolMarker);
                var unicodeSymbolToCharConverter = new
                ↪  UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
                ↪  unicodeSymbolCriterionMatcher);

                var unicodeSequenceCriterionMatcher = new TargetMatcher<ulong>(links,
                ↪  unicodeSequenceMarker);

                var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
                ↪  unicodeSymbolCriterionMatcher.IsMatched);

                var unicodeSequenceToStringConverter = new
                ↪  UnicodeSequenceToStringConverter<ulong>(links,
                ↪  unicodeSequenceCriterionMatcher, sequenceWalker,
                ↪  unicodeSymbolToCharConverter);

                var resultingString =
                ↪  unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);

                Assert.Equal(originalString, resultingString);
            }
        }
    }
}
```

Index