# 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
```

```
17        {
18            [MethodImpl(MethodImplOptions.AggressiveInlining)]
19            get => true;
20        }
21    }
22
23    protected readonly DisposableWithMultipleCallsAllowed Disposable;
24
25    [MethodImpl(MethodImplOptions.AggressiveInlining)]
26    protected LinksDisposableDecoratorBase(ILinks<TLink> links) : base(links) => Disposable
   ↪   = new DisposableWithMultipleCallsAllowed(Dispose);
27
28    [MethodImpl(MethodImplOptions.AggressiveInlining)]
29    ~LinksDisposableDecoratorBase() => Disposable.Destruct();
30
31    [MethodImpl(MethodImplOptions.AggressiveInlining)]
32    public void Dispose() => Disposable.Dispose();
33
34    [MethodImpl(MethodImplOptions.AggressiveInlining)]
35    protected virtual void Dispose(bool manual, bool wasDisposed)
36    {
37        if (!wasDisposed)
38        {
39            _links.DisposeIfPossible();
40        }
41    }
42    }
43 }
```

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Decorators
8  {
9      // TODO: Make LinksExternalReferenceValidator. A layer that checks each link to exist or to
   ↪   be external (hybrid link's raw number).
10     public class LinksInnerReferenceExistenceValidator<TLink> : LinksDecoratorBase<TLink>
11     {
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public LinksInnerReferenceExistenceValidator(ILinks<TLink> links) : base(links) { }
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
17         {
18             var links = _links;
19             links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
20             return links.Each(handler, restrictions);
21         }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
25         {
26             // TODO: Possible values: null, ExistentLink or NonExistentHybrid(ExternalReference)
27             var links = _links;
28             links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
29             links.EnsureInnerReferenceExists(substitution, nameof(substitution));
30             return links.Update(restrictions, substitution);
31         }
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public override void Delete(IList<TLink> restrictions)
35         {
36             var link = restrictions[_constants.IndexPart];
37             var links = _links;
38             links.EnsureLinkExists(link, nameof(link));
39             links.Delete(link);
40         }
41     }
42 }
```

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
```

```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)]
```

```csharp
            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

```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 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

```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 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

```csharp
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Decorators
7  {
8      public class LinksUsagesValidator<TLink> : LinksDecoratorBase<TLink>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public LinksUsagesValidator(ILinks<TLink> links) : base(links) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
15         {
16             var links = _links;
17             links.EnsureNoUsages(restrictions[_constants.IndexPart]);
18             return links.Update(restrictions, substitution);
19         }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public override void Delete(IList<TLink> restrictions)
23         {
24             var link = restrictions[_constants.IndexPart];
25             var links = _links;
26             links.EnsureNoUsages(link);
27             links.Delete(link);
28         }
29     }
30 }
```

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

```csharp
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Decorators
7  {
8      public class NonNullContentsLinkDeletionResolver<TLink> : LinksDecoratorBase<TLink>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public NonNullContentsLinkDeletionResolver(ILinks<TLink> links) : base(links) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public override void Delete(IList<TLink> restrictions)
15         {
16             var linkIndex = restrictions[_constants.IndexPart];
17             var links = _links;
18             links.EnforceResetValues(linkIndex);
19             links.Delete(linkIndex);
20         }
21     }
22 }
```

## 1.14 ./csharp/Platform.Data.Doublets/Decorators/UInt32Links.cs

```csharp
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using TLink = System.UInt32;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Decorators
8  {
9      public class UInt32Links : LinksDisposableDecoratorBase<TLink>
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public UInt32Links(ILinks<TLink> links) : base(links) { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public override TLink Create(IList<TLink> restrictions) => _links.CreatePoint();
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
19         {
20             var constants = _constants;
21             var indexPartConstant = constants.IndexPart;
22             var sourcePartConstant = constants.SourcePart;
23             var targetPartConstant = constants.TargetPart;
```

```csharp
                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<TLink> 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/UInt64Links.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
{
    /// <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>
```

```csharp
    {
        [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);
        }
    }
}
```

## 1.16   ./csharp/Platform.Data.Doublets/Decorators/UniLinks.cs

```csharp
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>
    {
```

```csharp
        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;
```

```
163              //      }
164              //      if (!matchedLinks.IsNullOrEmpty())
165              //      {
166              //          var totalMatchedLinks = matchedLinks.Count;
167              //          for (var i = 0; i < totalMatchedLinks; i++)
168              //          {
169              //              var matchedLink = matchedLinks[i];
170              //              if (substitutedHandler != null)
171              //              {
172              //                  var newValue = new List<T>(); // TODO: Prepare value to update here
173              //                  // TODO: Decide is it actually needed to use Before and After
     ↪  substitution handling.
174              //                  var substitutedDecision = substitutedHandler(matchedLink,
     ↪  newValue);
175              //                  if (Equals(substitutedDecision, Constants.Break))
176              //                      return Constants.Break;
177              //                  if (Equals(substitutedDecision, Constants.Continue))
178              //                  {
179              //                      // Actual update here
180              //                      Memory.SetLinkValue(newValue);
181              //                  }
182              //                  if (Equals(substitutedDecision, Constants.Skip))
183              //                  {
184              //                      // Cancel the update. TODO: decide use separate Cancel
     ↪  constant or Skip is enough?
185              //                  }
186              //              }
187              //          }
188              //      }
189              //}
190              return _constants.Continue;
191          }
192
193          public TLink Trigger(IList<TLink> patternOrCondition, Func<IList<TLink>, TLink>
     ↪  matchHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
     ↪  substitutionHandler)
194          {
195              var constants = _constants;
196              if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
197              {
198                  return constants.Continue;
199              }
200              else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO:
     ↪  Check if it is a correct condition
201              {
202                  // Or it only applies to trigger without matchHandler.
203                  throw new NotImplementedException();
204              }
205              else if (!substitution.IsNullOrEmpty()) // Creation
206              {
207                  var before = Array.Empty<TLink>();
208                  // Что должно означать False здесь? Остановиться (перестать идти) или пропустить
     ↪  (пройти мимо) или пустить (взять)?
209                  if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
     ↪  constants.Break))
210                  {
211                      return constants.Break;
212                  }
213                  var after = (IList<TLink>)substitution.ToArray();
214                  if (_equalityComparer.Equals(after[0], default))
215                  {
216                      var newLink = _links.Create();
217                      after[0] = newLink;
218                  }
219                  if (substitution.Count == 1)
220                  {
221                      after = _links.GetLink(substitution[0]);
222                  }
223                  else if (substitution.Count == 3)
224                  {
225                      //Links.Create(after);
226                  }
227                  else
228                  {
229                      throw new NotSupportedException();
230                  }
231                  if (matchHandler != null)
232                  {
```

```
233                         return substitutionHandler(before, after);
234                     }
235                     return constants.Continue;
236                 }
237             else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
238             {
239                 if (patternOrCondition.Count == 1)
240                 {
241                     var linkToDelete = patternOrCondition[0];
242                     var before = _links.GetLink(linkToDelete);
243                     if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                    ↪  constants.Break))
244                     {
245                         return constants.Break;
246                     }
247                     var after = Array.Empty<TLink>();
248                     _links.Update(linkToDelete, constants.Null, constants.Null);
249                     _links.Delete(linkToDelete);
250                     if (matchHandler != null)
251                     {
252                         return substitutionHandler(before, after);
253                     }
254                     return constants.Continue;
255                 }
256                 else
257                 {
258                     throw new NotSupportedException();
259                 }
260             }
261             else // Replace / Update
262             {
263                 if (patternOrCondition.Count == 1) //-V3125
264                 {
265                     var linkToUpdate = patternOrCondition[0];
266                     var before = _links.GetLink(linkToUpdate);
267                     if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                    ↪  constants.Break))
268                     {
269                         return constants.Break;
270                     }
271                     var after = (IList<TLink>)substitution.ToArray(); //-V3125
272                     if (_equalityComparer.Equals(after[0], default))
273                     {
274                         after[0] = linkToUpdate;
275                     }
276                     if (substitution.Count == 1)
277                     {
278                         if (!_equalityComparer.Equals(substitution[0], linkToUpdate))
279                         {
280                             after = _links.GetLink(substitution[0]);
281                             _links.Update(linkToUpdate, constants.Null, constants.Null);
282                             _links.Delete(linkToUpdate);
283                         }
284                     }
285                     else if (substitution.Count == 3)
286                     {
287                         //Links.Update(after);
288                     }
289                     else
290                     {
291                         throw new NotSupportedException();
292                     }
293                     if (matchHandler != null)
294                     {
295                         return substitutionHandler(before, after);
296                     }
297                     return constants.Continue;
298                 }
299                 else
300                 {
301                     throw new NotSupportedException();
302                 }
303             }
304         }
305
306     /// <remarks>
307     /// IList[IList[IList[T]]]
308     /// |       |       |      |||
```

```
309    ///  |      |         ------  ||
310    ///  |      |           link  ||
311    ///  |      ------------- |
312    ///  |          change       |
313    ///   --------------------
314    ///        changes
315    /// </remarks>
316    public IList<IList<IList<TLink>>> Trigger(IList<TLink> condition, IList<TLink>
       ↪  substitution)
317    {
318        var changes = new List<IList<IList<TLink>>>();
319        var @continue = _constants.Continue;
320        Trigger(condition, AlwaysContinue, substitution, (before, after) =>
321        {
322            var change = new[] { before, after };
323            changes.Add(change);
324            return @continue;
325        });
326        return changes;
327    }

329    private TLink AlwaysContinue(IList<TLink> linkToMatch) => _constants.Continue;
330    }
331 }
```

## 1.17   ./csharp/Platform.Data.Doublets/Doublet.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets
8  {
9
10     /// <summury>
11     /// <para>.</para>
12     /// <para>.</para>
13     /// </summury>
14     /// <typeparam>
15     /// <para>.</para>
16     /// <para>.</para>
17     /// </typeparam>
18     public struct Doublet<T> : IEquatable<Doublet<T>>
19     {
20         private static readonly EqualityComparer<T> _equalityComparer =
           ↪  EqualityComparer<T>.Default;
21
22         /// <summury>
23         /// <para>.</para>
24         /// <para>.</para>
25         /// </summury>
26         /// <typeparam name="T">
27         /// <para>.</para>
28         /// <para>.</para>
29         /// </typeparam>
30         public readonly T Source;
31
32         /// <summury>
33         /// <para>.</para>
34         /// <para>.</para>
35         /// </summury>
36         /// <typeparam name="T">
37         /// <para>.</para>
38         /// <para>.</para>
39         /// </typeparam>
40         public readonly T Target;
41
42         /// <summury>
43         /// <para>.</para>
44         /// <para>.</para>
45         /// </summury>
46         /// <typeparam name="T">
47         /// <para>.</para>
48         /// <para>.</para>
49         /// </typeparam>
50         /// <param name="source">
51         /// <para>.</para>
52         /// <para>.</para>
```

```csharp
        /// </param>
        /// <param name="target">
        /// <para>.</para>
        /// <para>.</para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Doublet(T source, T target)
        {
            Source = source;
            Target = target;
        }

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

        /// <summury>
        /// <para>.</para>
        /// <para>.</para>
        /// </summury>
        /// <typeparam>
        /// <para>.</para>
        /// <para>.</para>
        /// </typeparam>
        /// <param name="other">
        /// <para>.</para>
        /// <para>.</para>
        /// </param>
        /// <returns>
        /// <para>.</para>
        /// <para>.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Equals(Doublet<T> other) => _equalityComparer.Equals(Source, other.Source)
            && _equalityComparer.Equals(Target, other.Target);

        /// <summury>
        /// <para>.</para>
        /// <para>.</para>
        /// </summury>
        /// <typeparam>
        /// <para>.</para>
        /// <para>.</para>
        /// </typeparam>
        /// <param name="obj">
        /// <para>.</para>
        /// <para>.</para>
        /// </param>
        /// <returns>
        /// <para>.</para>
        /// <para>.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Equals(object obj) => obj is Doublet<T> doublet ?
            base.Equals(doublet) : false;

        /// <summury>
        /// <para>.</para>
        /// <para>.</para>
        /// </summury>
        /// <returns>
        /// <para>.</para>
        /// <para>.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override int GetHashCode() => (Source, Target).GetHashCode();

        /// <summury>
        /// <para>.</para>
        /// <para>.</para>
        /// </summury>
```

```
129        /// <param name="left">
130        /// <para>.</para>
131        /// <para>.</para>
132        /// </param>
133        /// <param name="right">
134        /// <para>.</para>
135        /// <para>.</para>
136        /// </param>
137        /// <returns>
138        /// <para>.</para>
139        /// <para>.</para>
140        /// </returns>
141        [MethodImpl(MethodImplOptions.AggressiveInlining)]
142        public static bool operator ==(Doublet<T> left, Doublet<T> right) => left.Equals(right);
143
144        /// <summury>
145        /// <para>.</para>
146        /// <para>.</para>
147        /// </summury>
148        /// <param name="left">
149        /// <para>.</para>
150        /// <para>.</para>
151        /// </param>
152        /// <param name="right">
153        /// <para>.</para>
154        /// <para>.</para>
155        /// </param>
156        /// <returns>
157        /// <para>.</para>
158        /// <para>.</para>
159        /// </returns>
160        [MethodImpl(MethodImplOptions.AggressiveInlining)]
161        public static bool operator !=(Doublet<T> left, Doublet<T> right) => !(left == right);
162    }
163 }
```

## 1.18   ./csharp/Platform.Data.Doublets/DoubletComparer.cs

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets
7  {
8      /// <remarks>
9      /// TODO: Может стоит попробовать ref во всех методах (IRefEqualityComparer)
10     /// 2x faster with comparer
11     /// </remarks>
12     public class DoubletComparer<T> : IEqualityComparer<Doublet<T>>
13     {
14         public static readonly DoubletComparer<T> Default = new DoubletComparer<T>();
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public bool Equals(Doublet<T> x, Doublet<T> y) => x.Equals(y);
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public int GetHashCode(Doublet<T> obj) => obj.GetHashCode();
21     }
22 }
```

## 1.19   ./csharp/Platform.Data.Doublets/ILinks.cs

```
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  using System.Collections.Generic;
4
5  namespace Platform.Data.Doublets
6  {
7      public interface ILinks<TLink> : ILinks<TLink, LinksConstants<TLink>>
8      {
9      }
10 }
```

## 1.20   ./csharp/Platform.Data.Doublets/ILinksExtensions.cs

```
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System.Linq;
5  using System.Runtime.CompilerServices;
6  using Platform.Ranges;
```

```csharp
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));
```

```csharp
        /// <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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IList<TLink> SingleOrDefault<TLink>(this ILinks<TLink> links, IList<TLink>
        ↪  query)
        {
            IList<TLink> result = null;
            var count = 0;
            var constants = links.Constants;
            var @continue = constants.Continue;
            var @break = constants.Break;
            links.Each(linkHandler, query);
            return result;

            TLink linkHandler(IList<TLink> link)
            {
                if (count == 0)
                {
                    result = link;
                    count++;
                    return @continue;
                }
                else
                {
                    result = null;
                    return @break;
                }
            }
        }

        #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
```

```csharp
        /// 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;
```

```csharp
            }

        #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.
```

```csharp
        /// </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,
            ↪  ulong>.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,
            ↪  ulong>.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>();
            }
        }
```

```csharp
        /// <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);
            }
        }
```

```csharp
        }

        [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.Max⌋
                ↪  imum)));
                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
```

```csharp
473
474            /// <param name="links">Хранилище связей.</param>
475            [MethodImpl(MethodImplOptions.AggressiveInlining)]
476            public static TLink CountUsages<TLink>(this ILinks<TLink> links, TLink link)
477            {
478                var constants = links.Constants;
479                var values = links.GetLink(link);
480                TLink usagesAsSource = links.Count(new Link<TLink>(constants.Any, link,
                    ↪   constants.Any));
481                var equalityComparer = EqualityComparer<TLink>.Default;
482                if (equalityComparer.Equals(values[constants.SourcePart], link))
483                {
484                    usagesAsSource = Arithmetic<TLink>.Decrement(usagesAsSource);
485                }
486                TLink usagesAsTarget = links.Count(new Link<TLink>(constants.Any, constants.Any,
                    ↪   link));
487                if (equalityComparer.Equals(values[constants.TargetPart], link))
488                {
489                    usagesAsTarget = Arithmetic<TLink>.Decrement(usagesAsTarget);
490                }
491                return Arithmetic<TLink>.Add(usagesAsSource, usagesAsTarget);
492            }
493
494            /// <param name="links">Хранилище связей.</param>
495            [MethodImpl(MethodImplOptions.AggressiveInlining)]
496            public static bool HasUsages<TLink>(this ILinks<TLink> links, TLink link) =>
                ↪   Comparer<TLink>.Default.Compare(links.CountUsages(link), default) > 0;
497
498            /// <param name="links">Хранилище связей.</param>
499            [MethodImpl(MethodImplOptions.AggressiveInlining)]
500            public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source,
                ↪   TLink target)
501            {
502                var constants = links.Constants;
503                var values = links.GetLink(link);
504                var equalityComparer = EqualityComparer<TLink>.Default;
505                return equalityComparer.Equals(values[constants.SourcePart], source) &&
                    ↪   equalityComparer.Equals(values[constants.TargetPart], target);
506            }
507
508            /// <summary>
509            /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
510            /// </summary>
511            /// <param name="links">Хранилище связей.</param>
512            /// <param name="source">Индекс связи, которая является началом для искомой
                ↪   связи.</param>
513            /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
514            /// <returns>Индекс искомой связи с указанными Source (началом) и Target
                ↪   (концом).</returns>
515            [MethodImpl(MethodImplOptions.AggressiveInlining)]
516            public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
                ↪   target)
517            {
518                var contants = links.Constants;
519                var setter = new Setter<TLink, TLink>(contants.Continue, contants.Break, default);
520                links.Each(setter.SetFirstAndReturnFalse, contants.Any, source, target);
521                return setter.Result;
522            }
523
524            /// <param name="links">Хранилище связей.</param>
525            [MethodImpl(MethodImplOptions.AggressiveInlining)]
526            public static TLink Create<TLink>(this ILinks<TLink> links) => links.Create(null);
527
528            /// <param name="links">Хранилище связей.</param>
529            [MethodImpl(MethodImplOptions.AggressiveInlining)]
530            public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
531            {
532                var link = links.Create();
533                return links.Update(link, link, link);
534            }
535
536            /// <param name="links">Хранилище связей.</param>
537            [MethodImpl(MethodImplOptions.AggressiveInlining)]
538            public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
                ↪   target) => links.Update(links.Create(), source, target);
539
540            /// <summary>
541            /// Обновляет связь с указанными началом (Source) и концом (Target)
```

```csharp
542         /// на связь с указанными началом (NewSource) и концом (NewTarget).
543         /// </summary>
544         /// <param name="links">Хранилище связей.</param>
545         /// <param name="link">Индекс обновляемой связи.</param>
546         /// <param name="newSource">Индекс связи, которая является началом связи, на которую
               ↪   выполняется обновление.</param>
547         /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
               ↪   выполняется обновление.</param>
548         /// <returns>Индекс обновлённой связи.</returns>
549         [MethodImpl(MethodImplOptions.AggressiveInlining)]
550         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));

551
552         /// <summary>
553         /// Обновляет связь с указанными началом (Source) и концом (Target)
554         /// на связь с указанными началом (NewSource) и концом (NewTarget).
555         /// </summary>
556         /// <param name="links">Хранилище связей.</param>
557         /// <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>
558         /// <returns>Индекс обновлённой связи.</returns>
559         [MethodImpl(MethodImplOptions.AggressiveInlining)]
560         public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
561         {
562             if (restrictions.Length == 2)
563             {
564                 return links.MergeAndDelete(restrictions[0], restrictions[1]);
565             }
566             if (restrictions.Length == 4)
567             {
568                 return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
                       ↪   restrictions[2], restrictions[3]);
569             }
570             else
571             {
572                 return links.Update(new LinkAddress<TLink>(restrictions[0]), restrictions);
573             }
574         }

575
576         [MethodImpl(MethodImplOptions.AggressiveInlining)]
577         public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
               ↪   links, TLink constant, IList<TLink> restrictions, IList<TLink> substitution)
578         {
579             var equalityComparer = EqualityComparer<TLink>.Default;
580             var constants = links.Constants;
581             var restrictionsIndex = restrictions[constants.IndexPart];
582             var substitutionIndex = substitution[constants.IndexPart];
583             if (equalityComparer.Equals(substitutionIndex, default))
584             {
585                 substitutionIndex = restrictionsIndex;
586             }
587             var source = substitution[constants.SourcePart];
588             var target = substitution[constants.TargetPart];
589             source = equalityComparer.Equals(source, constant) ? substitutionIndex : source;
590             target = equalityComparer.Equals(target, constant) ? substitutionIndex : target;
591             return new Link<TLink>(substitutionIndex, source, target);
592         }

593
594         /// <summary>
595         /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
               ↪   с указанными Source (началом) и Target (концом).
596         /// </summary>
597         /// <param name="links">Хранилище связей.</param>
598         /// <param name="source">Индекс связи, которая является началом на создаваемой
               ↪   связи.</param>
599         /// <param name="target">Индекс связи, которая является концом для создаваемой
               ↪   связи.</param>
600         /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
601         [MethodImpl(MethodImplOptions.AggressiveInlining)]
602         public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
               ↪   target)
603         {
604             var link = links.SearchOrDefault(source, target);
605             if (EqualityComparer<TLink>.Default.Equals(link, default))
```

```csharp
                {
                    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);
            }
```

```
676
677          [MethodImpl(MethodImplOptions.AggressiveInlining)]
678          public static void DeleteByQuery<TLink>(this ILinks<TLink> links, Link<TLink> query)
679          {
680              var count = CheckedConverter<TLink, long>.Default.Convert(links.Count(query));
681              if (count > 0)
682              {
683                  var queryResult = new TLink[count];
684                  var queryResultFiller = new ArrayFiller<TLink, TLink>(queryResult,
                     ↪  links.Constants.Continue);
685                  links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
686                  for (var i = count - 1; i >= 0; i--)
687                  {
688                      links.Delete(queryResult[i]);
689                  }
690              }
691          }
692
693          // TODO: Move to Platform.Data
694          [MethodImpl(MethodImplOptions.AggressiveInlining)]
695          public static bool AreValuesReset<TLink>(this ILinks<TLink> links, TLink linkIndex)
696          {
697              var nullConstant = links.Constants.Null;
698              var equalityComparer = EqualityComparer<TLink>.Default;
699              var link = links.GetLink(linkIndex);
700              for (int i = 1; i < link.Count; i++)
701              {
702                  if (!equalityComparer.Equals(link[i], nullConstant))
703                  {
704                      return false;
705                  }
706              }
707              return true;
708          }
709
710          // TODO: Create a universal version of this method in Platform.Data (with using of for
             ↪  loop)
711          [MethodImpl(MethodImplOptions.AggressiveInlining)]
712          public static void ResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
713          {
714              var nullConstant = links.Constants.Null;
715              var updateRequest = new Link<TLink>(linkIndex, nullConstant, nullConstant);
716              links.Update(updateRequest);
717          }
718
719          // TODO: Create a universal version of this method in Platform.Data (with using of for
             ↪  loop)
720          [MethodImpl(MethodImplOptions.AggressiveInlining)]
721          public static void EnforceResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
722          {
723              if (!links.AreValuesReset(linkIndex))
724              {
725                  links.ResetValues(linkIndex);
726              }
727          }
728
729          /// <summary>
730          /// Merging two usages graphs, all children of old link moved to be children of new link
             ↪  or deleted.
731          /// </summary>
732          [MethodImpl(MethodImplOptions.AggressiveInlining)]
733          public static TLink MergeUsages<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
             ↪  TLink newLinkIndex)
734          {
735              var addressToInt64Converter = CheckedConverter<TLink, long>.Default;
736              var equalityComparer = EqualityComparer<TLink>.Default;
737              if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
738              {
739                  var constants = links.Constants;
740                  var usagesAsSourceQuery = new Link<TLink>(constants.Any, oldLinkIndex,
                     ↪  constants.Any);
741                  var usagesAsSourceCount =
                     ↪  addressToInt64Converter.Convert(links.Count(usagesAsSourceQuery));
742                  var usagesAsTargetQuery = new Link<TLink>(constants.Any, constants.Any,
                     ↪  oldLinkIndex);
743                  var usagesAsTargetCount =
                     ↪  addressToInt64Converter.Convert(links.Count(usagesAsTargetQuery));
```

```csharp
                    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]})";
        }
```

```
814
815        [MethodImpl(MethodImplOptions.AggressiveInlining)]
816        public static string Format<TLink>(this ILinks<TLink> links, TLink link) =>
           ↪  links.Format(links.GetLink(link));
817    }
818 }
```

## 1.21 ./csharp/Platform.Data.Doublets/ISynchronizedLinks.cs

```
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Data.Doublets
4  {
5      public interface ISynchronizedLinks<TLink> : ISynchronizedLinks<TLink, ILinks<TLink>,
        ↪  LinksConstants<TLink>>, ILinks<TLink>
6      {
7      }
8  }
```

## 1.22 ./csharp/Platform.Data.Doublets/Incrementers/FrequencyIncrementer.cs

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Incrementers;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Incrementers
8  {
9      public class FrequencyIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
12
13         private readonly TLink _frequencyMarker;
14         private readonly TLink _unaryOne;
15         private readonly IIncrementer<TLink> _unaryNumberIncrementer;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
           ↪  IIncrementer<TLink> unaryNumberIncrementer)
19             : base(links)
20         {
21             _frequencyMarker = frequencyMarker;
22             _unaryOne = unaryOne;
23             _unaryNumberIncrementer = unaryNumberIncrementer;
24         }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public TLink Increment(TLink frequency)
28         {
29             var links = _links;
30             if (_equalityComparer.Equals(frequency, default))
31             {
32                 return links.GetOrCreate(_unaryOne, _frequencyMarker);
33             }
34             var incrementedSource =
               ↪  _unaryNumberIncrementer.Increment(links.GetSource(frequency));
35             return links.GetOrCreate(incrementedSource, _frequencyMarker);
36         }
37     }
38 }
```

## 1.23 ./csharp/Platform.Data.Doublets/Incrementers/UnaryNumberIncrementer.cs

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Incrementers;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Incrementers
8  {
9      public class UnaryNumberIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
12
13         private readonly TLink _unaryOne;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) : base(links) =>
           ↪  _unaryOne = unaryOne;
```

```csharp
        [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));
            }
        }
    }
}
```

## 1.24   ./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)]
```

```csharp
54          public Link(ref Link<TLink> other) => SetValues(ref other, out Index, out Source, out
    ↪   Target);

56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          public Link(TLink index, TLink source, TLink target)
58          {
59              Index = index;
60              Source = source;
61              Target = target;
62          }

64          [MethodImpl(MethodImplOptions.AggressiveInlining)]
65          private static void SetValues(ref Link<TLink> other, out TLink index, out TLink source,
    ↪   out TLink target)
66          {
67              index = other.Index;
68              source = other.Source;
69              target = other.Target;
70          }

72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          private static void SetValues(IList<TLink> values, out TLink index, out TLink source,
    ↪   out TLink target)
74          {
75              switch (values.Count)
76              {
77                  case 3:
78                      index = values[0];
79                      source = values[1];
80                      target = values[2];
81                      break;
82                  case 2:
83                      index = values[0];
84                      source = values[1];
85                      target = default;
86                      break;
87                  case 1:
88                      index = values[0];
89                      source = default;
90                      target = default;
91                      break;
92                  default:
93                      index = default;
94                      source = default;
95                      target = default;
96                      break;
97              }
98          }

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

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

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

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

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

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

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

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

```

```csharp
        [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;
        }
```

```csharp
            [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
        }
    }
```

## 1.25   ./csharp/Platform.Data.Doublets/LinkExtensions.cs

```csharp
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);
    }
}
```

## 1.26   ./csharp/Platform.Data.Doublets/LinksOperatorBase.cs

```csharp
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;
    }
}
```

## 1.27 ./csharp/Platform.Data.Doublets/Memory/ILinksListMethods.cs

```csharp
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);
    }
}
```

## 1.28 ./csharp/Platform.Data.Doublets/Memory/ILinksTreeMethods.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.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);
    }
}
```

## 1.29 ./csharp/Platform.Data.Doublets/Memory/IndexTreeType.cs

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

namespace Platform.Data.Doublets.Memory
{
    public enum IndexTreeType
    {
        Default = 0,
        SizeBalancedTree = 1,
        RecursionlessSizeBalancedTree = 2,
        SizedAndThreadedAVLBalancedTree = 3
    }
}
```

## 1.30 ./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;
```

```csharp
            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);
        }
    }
```

## 1.31 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksRecursionlessSizeBalancedTreeMetho

```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 ExternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink> :
    ↪  RecursionlessSizeBalancedTreeMethods<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 ExternalLinksRecursionlessSizeBalancedTreeMethodsBase(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);
```

```csharp
        [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>
```

```csharp
        /// <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;
```

```
184                 if (EqualToZero(link))
185                 {
186                     return @continue;
187                 }
188                 var linkBasePart = GetBasePartValue(link);
189                 var @break = Break;
190                 if (GreaterThan(linkBasePart, @base))
191                 {
192                     if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
193                     {
194                         return @break;
195                     }
196                 }
197                 else if (LessThan(linkBasePart, @base))
198                 {
199                     if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
200                     {
201                         return @break;
202                     }
203                 }
204                 else //if (linkBasePart == @base)
205                 {
206                     if (AreEqual(handler(GetLinkValues(link)), @break))
207                     {
208                         return @break;
209                     }
210                     if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
211                     {
212                         return @break;
213                     }
214                     if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
215                     {
216                         return @break;
217                     }
218                 }
219                 return @continue;
220             }
221
222         [MethodImpl(MethodImplOptions.AggressiveInlining)]
223         protected override void PrintNodeValue(TLink node, StringBuilder sb)
224         {
225             ref var link = ref GetLinkDataPartReference(node);
226             sb.Append(' ');
227             sb.Append(link.Source);
228             sb.Append('-');
229             sb.Append('>');
230             sb.Append(link.Target);
231         }
232     }
233 }
```

## 1.32   ./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSizeBalancedTreeMethodsBase.cs

```
1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Converters;
7  using static System.Runtime.CompilerServices.Unsafe;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets.Memory.Split.Generic
12 {
13     public unsafe abstract class ExternalLinksSizeBalancedTreeMethodsBase<TLink> :
     ↪   SizeBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
14     {
15         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
     ↪       UncheckedConverter<TLink, long>.Default;
16
17         protected readonly TLink Break;
18         protected readonly TLink Continue;
19         protected readonly byte* LinksDataParts;
20         protected readonly byte* LinksIndexParts;
21         protected readonly byte* Header;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected ExternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants,
     ↪       byte* linksDataParts, byte* linksIndexParts, byte* header)
25         {
```

```csharp
                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;
                        }
```

```
 96                        if (AreEqual(index, leftSize))
 97                        {
 98                            return root;
 99                        }
100                        root = GetRightOrDefault(root);
101                        index = Subtract(index, Increment(leftSize));
102                    }
103                    return Zero; // TODO: Impossible situation exception (only if tree structure
                        ↪ broken)
104                }
105            }
106
107            /// <summary>
108            /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
                ↪ (концом).
109            /// </summary>
110            /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
111            /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
112            /// <returns>Индекс искомой связи.</returns>
113            [MethodImpl(MethodImplOptions.AggressiveInlining)]
114            public TLink Search(TLink source, TLink target)
115            {
116                var root = GetTreeRoot();
117                while (!EqualToZero(root))
118                {
119                    ref var rootLink = ref GetLinkDataPartReference(root);
120                    var rootSource = rootLink.Source;
121                    var rootTarget = rootLink.Target;
122                    if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                        ↪ node.Key < root.Key
123                    {
124                        root = GetLeftOrDefault(root);
125                    }
126                    else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
                        ↪ node.Key > root.Key
127                    {
128                        root = GetRightOrDefault(root);
129                    }
130                    else // node.Key == root.Key
131                    {
132                        return root;
133                    }
134                }
135                return Zero;
136            }
137
138            // TODO: Return indices range instead of references count
139            [MethodImpl(MethodImplOptions.AggressiveInlining)]
140            public TLink CountUsages(TLink link)
141            {
142                var root = GetTreeRoot();
143                var total = GetSize(root);
144                var totalRightIgnore = Zero;
145                while (!EqualToZero(root))
146                {
147                    var @base = GetBasePartValue(root);
148                    if (LessOrEqualThan(@base, link))
149                    {
150                        root = GetRightOrDefault(root);
151                    }
152                    else
153                    {
154                        totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
155                        root = GetLeftOrDefault(root);
156                    }
157                }
158                root = GetTreeRoot();
159                var totalLeftIgnore = Zero;
160                while (!EqualToZero(root))
161                {
162                    var @base = GetBasePartValue(root);
163                    if (GreaterOrEqualThan(@base, link))
164                    {
165                        root = GetLeftOrDefault(root);
166                    }
167                    else
168                    {
169                        totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
```

```
170                         root = GetRightOrDefault(root);
171                     }
172                 }
173                 return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
174             }
175
176             [MethodImpl(MethodImplOptions.AggressiveInlining)]
177             public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
        ↪     EachUsageCore(@base, GetTreeRoot(), handler);
178
179             // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
        ↪     low-level MSIL stack.
180             [MethodImpl(MethodImplOptions.AggressiveInlining)]
181             private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
182             {
183                 var @continue = Continue;
184                 if (EqualToZero(link))
185                 {
186                     return @continue;
187                 }
188                 var linkBasePart = GetBasePartValue(link);
189                 var @break = Break;
190                 if (GreaterThan(linkBasePart, @base))
191                 {
192                     if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
193                     {
194                         return @break;
195                     }
196                 }
197                 else if (LessThan(linkBasePart, @base))
198                 {
199                     if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
200                     {
201                         return @break;
202                     }
203                 }
204                 else //if (linkBasePart == @base)
205                 {
206                     if (AreEqual(handler(GetLinkValues(link)), @break))
207                     {
208                         return @break;
209                     }
210                     if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
211                     {
212                         return @break;
213                     }
214                     if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
215                     {
216                         return @break;
217                     }
218                 }
219                 return @continue;
220             }
221
222             [MethodImpl(MethodImplOptions.AggressiveInlining)]
223             protected override void PrintNodeValue(TLink node, StringBuilder sb)
224             {
225                 ref var link = ref GetLinkDataPartReference(node);
226                 sb.Append(' ');
227                 sb.Append(link.Source);
228                 sb.Append('-');
229                 sb.Append('>');
230                 sb.Append(link.Target);
231             }
232         }
233     }
```

## 1.33  ./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSourcesRecursionlessSizeBalancedTree

```
1   using System.Runtime.CompilerServices;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Data.Doublets.Memory.Split.Generic
6   {
7       public unsafe class ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods<TLink> :
        ↪     ExternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
8       {
9           [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
        public ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(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;
        }
    }
}
```

## 1.34 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSourcesSizeBalancedTreeMethods.cs

```csharp
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) { }
```

```csharp
            [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;
            }
        }
    }
```

## 1.35 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksTargetsRecursionlessSizeBalancedTree

```csharp
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 ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods<TLink> :
    ↪   ExternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
        ↪   constants, byte* linksDataParts, byte* linksIndexParts, byte* header) :
        ↪   base(constants, linksDataParts, linksIndexParts, header) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
        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;
        }
    }
}
```

## 1.36 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksTargetsSizeBalancedTreeMethods.cs

```csharp
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;
```

```csharp
        [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;
        }
    }
}
```

## 1.37 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksRecursionlessSizeBalancedTreeMethod

```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 InternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink> :
    ↪   RecursionlessSizeBalancedTreeMethods<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;
```

```csharp
        protected readonly byte* Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected InternalLinksRecursionlessSizeBalancedTreeMethodsBase(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;
                    }
                    root = GetRightOrDefault(root);
                    index = Subtract(index, Increment(leftSize));
                }
                return Zero; // TODO: Impossible situation exception (only if tree structure
                ↪ broken)
            }
        }

        /// <summary>
```

```csharp
        /// 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);
        }
    }
}
```

```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 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;
                }
            }
        }
```

```csharp
                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 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;
```

```
146         }
147         if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
148         {
149             return @break;
150         }
151         return @continue;
152     }
153
154     [MethodImpl(MethodImplOptions.AggressiveInlining)]
155     protected override void PrintNodeValue(TLink node, StringBuilder sb)
156     {
157         ref var link = ref GetLinkDataPartReference(node);
158         sb.Append(' ');
159         sb.Append(link.Source);
160         sb.Append('-');
161         sb.Append('>');
162         sb.Append(link.Target);
163     }
164 }
165 }
```

## 1.39  ./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSourcesLinkedListMethods.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Collections.Methods.Lists;
5  using Platform.Converters;
6  using static System.Runtime.CompilerServices.Unsafe;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Memory.Split.Generic
11 {
12     public unsafe class InternalLinksSourcesLinkedListMethods<TLink> :
     ↪  RelativeCircularDoublyLinkedListMethods<TLink>
13     {
14         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
         ↪  UncheckedConverter<TLink, long>.Default;
15         private readonly byte* _linksDataParts;
16         private readonly byte* _linksIndexParts;
17         protected readonly TLink Break;
18         protected readonly TLink Continue;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public InternalLinksSourcesLinkedListMethods(LinksConstants<TLink> constants, byte*
         ↪  linksDataParts, byte* linksIndexParts)
22         {
23             _linksDataParts = linksDataParts;
24             _linksIndexParts = linksIndexParts;
25             Break = constants.Break;
26             Continue = constants.Continue;
27         }
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
         ↪  AsRef<RawLinkDataPart<TLink>>(_linksDataParts + (RawLinkDataPart<TLink>.SizeInBytes
         ↪  * _addressToInt64Converter.Convert(link)));
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected virtual ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
         ↪  ref AsRef<RawLinkIndexPart<TLink>>(_linksIndexParts +
         ↪  (RawLinkIndexPart<TLink>.SizeInBytes * _addressToInt64Converter.Convert(link)));
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override TLink GetFirst(TLink head) =>
         ↪  GetLinkIndexPartReference(head).RootAsSource;
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         protected override TLink GetLast(TLink head)
40         {
41             var first = GetLinkIndexPartReference(head).RootAsSource;
42             if (EqualToZero(first))
43             {
44                 return first;
45             }
46             else
47             {
48                 return GetPrevious(first);
49             }
50         }
```

```csharp
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetPrevious(TLink element) =>
        ↪   GetLinkIndexPartReference(element).LeftAsSource;

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetFirst(TLink head, TLink element) =>
        ↪   GetLinkIndexPartReference(head).RootAsSource = element;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLast(TLink head, TLink element)
        {
            //var first = GetLinkIndexPartReference(head).RootAsSource;
            //if (EqualToZero(first))
            //{
            //    SetFirst(head, element);
            //}
            //else
            //{
            //    SetPrevious(first, element);
            //}
        }

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink CountUsages(TLink head) => GetSize(head);

        [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)]
        public TLink EachUsage(TLink source, Func<IList<TLink>, TLink> handler)
        {
            var @continue = Continue;
            var @break = Break;
            var current = GetFirst(source);
            var first = current;
            while (!EqualToZero(current))
            {
                if (AreEqual(handler(GetLinkValues(current)), @break))
                {
                    return @break;
                }
                current = GetNext(current);
                if (AreEqual(current, first))
                {
                    return @continue;
                }
            }
            return @continue;
        }
    }
}
```

```csharp
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 InternalLinksSourcesRecursionlessSizeBalancedTreeMethods<TLink> :
        InternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(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);
    }
}
```

```csharp
using System.Runtime.CompilerServices;

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

```

```csharp
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);
    }
}
```

## 1.42 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksTargetsRecursionlessSizeBalancedTree[

```csharp
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 InternalLinksTargetsRecursionlessSizeBalancedTreeMethods<TLink> :
    ↪   InternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
    {
```

```csharp
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(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);
    }
}
```

## 1.43  ./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksTargetsSizeBalancedTreeMethods.cs

```csharp
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) { }

```

```csharp
        [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);
    }
}
```

## 1.44   ./csharp/Platform.Data.Doublets/Memory/Split/Generic/SplitMemoryLinks.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.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;
```

```csharp
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          public SplitMemoryLinks(string dataMemory, string indexMemory) : this(new
    ↪   FileMappedResizableDirectMemory(dataMemory), new
    ↪   FileMappedResizableDirectMemory(indexMemory)) { }
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
    ↪   indexMemory) : this(dataMemory, indexMemory, DefaultLinksSizeStep) { }
26
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
    ↪   indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
    ↪   memoryReservationStep, Default<LinksConstants<TLink>>.Instance,
    ↪   IndexTreeType.Default, useLinkedList: true) { }
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
    ↪   indexMemory, long memoryReservationStep, LinksConstants<TLink> constants) :
    ↪   this(dataMemory, indexMemory, memoryReservationStep, constants,
    ↪   IndexTreeType.Default, useLinkedList: true) { }
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
    ↪   indexMemory, long memoryReservationStep, LinksConstants<TLink> constants,
    ↪   IndexTreeType indexTreeType, bool useLinkedList) : base(dataMemory, indexMemory,
    ↪   memoryReservationStep, constants, useLinkedList)
35          {
36              if (indexTreeType == IndexTreeType.SizeBalancedTree)
37              {
38                  _createInternalSourceTreeMethods = () => new
    ↪   InternalLinksSourcesSizeBalancedTreeMethods<TLink>(Constants,
    ↪   _linksDataParts, _linksIndexParts, _header);
39                  _createExternalSourceTreeMethods = () => new
    ↪   ExternalLinksSourcesSizeBalancedTreeMethods<TLink>(Constants,
    ↪   _linksDataParts, _linksIndexParts, _header);
40                  _createInternalTargetTreeMethods = () => new
    ↪   InternalLinksTargetsSizeBalancedTreeMethods<TLink>(Constants,
    ↪   _linksDataParts, _linksIndexParts, _header);
41                  _createExternalTargetTreeMethods = () => new
    ↪   ExternalLinksTargetsSizeBalancedTreeMethods<TLink>(Constants,
    ↪   _linksDataParts, _linksIndexParts, _header);
42              }
43              else
44              {
45                  _createInternalSourceTreeMethods = () => new
    ↪   InternalLinksSourcesRecursionlessSizeBalancedTreeMethods<TLink>(Constants,
    ↪   _linksDataParts, _linksIndexParts, _header);
46                  _createExternalSourceTreeMethods = () => new
    ↪   ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods<TLink>(Constants,
    ↪   _linksDataParts, _linksIndexParts, _header);
47                  _createInternalTargetTreeMethods = () => new
    ↪   InternalLinksTargetsRecursionlessSizeBalancedTreeMethods<TLink>(Constants,
    ↪   _linksDataParts, _linksIndexParts, _header);
48                  _createExternalTargetTreeMethods = () => new
    ↪   ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods<TLink>(Constants,
    ↪   _linksDataParts, _linksIndexParts, _header);
49              }
50              Init(dataMemory, indexMemory);
51          }
52
53          [MethodImpl(MethodImplOptions.AggressiveInlining)]
54          protected override void SetPointers(IResizableDirectMemory dataMemory,
    ↪   IResizableDirectMemory indexMemory)
55          {
56              _linksDataParts = (byte*)dataMemory.Pointer;
57              _linksIndexParts = (byte*)indexMemory.Pointer;
58              _header = _linksIndexParts;
59              if (_useLinkedList)
60              {
61                  InternalSourcesListMethods = new
    ↪   InternalLinksSourcesLinkedListMethods<TLink>(Constants, _linksDataParts,
    ↪   _linksIndexParts);
62              }
63              else
64              {
65                  InternalSourcesTreeMethods = _createInternalSourceTreeMethods();
66              }
```

```
67          ExternalSourcesTreeMethods = _createExternalSourceTreeMethods();
68          InternalTargetsTreeMethods = _createInternalTargetTreeMethods();
69          ExternalTargetsTreeMethods = _createExternalTargetTreeMethods();
70          UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_linksDataParts, _header);
71      }
72
73      [MethodImpl(MethodImplOptions.AggressiveInlining)]
74      protected override void ResetPointers()
75      {
76          base.ResetPointers();
77          _linksDataParts = null;
78          _linksIndexParts = null;
79          _header = null;
80      }
81
82      [MethodImpl(MethodImplOptions.AggressiveInlining)]
83      protected override ref LinksHeader<TLink> GetHeaderReference() => ref
    ↪   AsRef<LinksHeader<TLink>>(_header);
84
85      [MethodImpl(MethodImplOptions.AggressiveInlining)]
86      protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex)
    ↪   => ref AsRef<RawLinkDataPart<TLink>>(_linksDataParts + (LinkDataPartSizeInBytes *
    ↪   ConvertToInt64(linkIndex)));
87
88      [MethodImpl(MethodImplOptions.AggressiveInlining)]
89      protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
    ↪   linkIndex) => ref AsRef<RawLinkIndexPart<TLink>>(_linksIndexParts +
    ↪   (LinkIndexPartSizeInBytes * ConvertToInt64(linkIndex)));
90      }
91  }
```

## 1.45   ./csharp/Platform.Data.Doublets/Memory/Split/Generic/SplitMemoryLinksBase.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using Platform.Disposables;
5   using Platform.Singletons;
6   using Platform.Converters;
7   using Platform.Numbers;
8   using Platform.Memory;
9   using Platform.Data.Exceptions;
10
11  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13  namespace Platform.Data.Doublets.Memory.Split.Generic
14  {
15      public abstract class SplitMemoryLinksBase<TLink> : DisposableBase, ILinks<TLink>
16      {
17          private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪   EqualityComparer<TLink>.Default;
18          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
19          private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
        ↪   UncheckedConverter<TLink, long>.Default;
20          private static readonly UncheckedConverter<long, TLink> _int64ToAddressConverter =
        ↪   UncheckedConverter<long, TLink>.Default;
21
22          private static readonly TLink _zero = default;
23          private static readonly TLink _one = Arithmetic.Increment(_zero);
24
25          /// <summary>Возвращает размер одной связи в байтах.</summary>
26          /// <remarks>
27          /// Используется только во вне класса, не рекомедуется использовать внутри.
28          /// Так как во вне не обязательно будет доступен unsafe C#.
29          /// </remarks>
30          public static readonly long LinkDataPartSizeInBytes = RawLinkDataPart<TLink>.SizeInBytes;
31
32          public static readonly long LinkIndexPartSizeInBytes =
        ↪   RawLinkIndexPart<TLink>.SizeInBytes;
33
34          public static readonly long LinkHeaderSizeInBytes = LinksHeader<TLink>.SizeInBytes;
35
36          public static readonly long DefaultLinksSizeStep = 1 * 1024 * 1024;
37
38          protected readonly IResizableDirectMemory _dataMemory;
39          protected readonly IResizableDirectMemory _indexMemory;
40          protected readonly bool _useLinkedList;
41          protected readonly long _dataMemoryReservationStepInBytes;
42          protected readonly long _indexMemoryReservationStepInBytes;
43
44          protected InternalLinksSourcesLinkedListMethods<TLink> InternalSourcesListMethods;
45          protected ILinksTreeMethods<TLink> InternalSourcesTreeMethods;
```

```csharp
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, bool
    useLinkedList)
{
    _dataMemory = dataMemory;
    _indexMemory = indexMemory;
    _dataMemoryReservationStepInBytes = memoryReservationStep * LinkDataPartSizeInBytes;
    _indexMemoryReservationStepInBytes = memoryReservationStep *
        LinkIndexPartSizeInBytes;
    _useLinkedList = useLinkedList;
    Constants = constants;
}

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

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected virtual void Init(IResizableDirectMemory dataMemory, IResizableDirectMemory
    indexMemory)
{
    // Read allocated links from header
    if (indexMemory.ReservedCapacity < LinkHeaderSizeInBytes)
    {
        indexMemory.ReservedCapacity = LinkHeaderSizeInBytes;
    }
    SetPointers(dataMemory, indexMemory);
    ref var header = ref GetHeaderReference();
    var allocatedLinks = ConvertToInt64(header.AllocatedLinks);
    // Adjust reserved capacity
    var minimumDataReservedCapacity = allocatedLinks * LinkDataPartSizeInBytes;
    if (minimumDataReservedCapacity < dataMemory.UsedCapacity)
    {
        minimumDataReservedCapacity = dataMemory.UsedCapacity;
    }
    if (minimumDataReservedCapacity < _dataMemoryReservationStepInBytes)
    {
        minimumDataReservedCapacity = _dataMemoryReservationStepInBytes;
    }
    var minimumIndexReservedCapacity = allocatedLinks * LinkDataPartSizeInBytes;
    if (minimumIndexReservedCapacity < indexMemory.UsedCapacity)
    {
        minimumIndexReservedCapacity = indexMemory.UsedCapacity;
    }
    if (minimumIndexReservedCapacity < _indexMemoryReservationStepInBytes)
    {
        minimumIndexReservedCapacity = _indexMemoryReservationStepInBytes;
    }
    // Check for alignment
```

```csharp
116            if (minimumDataReservedCapacity % _dataMemoryReservationStepInBytes > 0)
117            {
118                minimumDataReservedCapacity = ((minimumDataReservedCapacity /
     ↪  _dataMemoryReservationStepInBytes) * _dataMemoryReservationStepInBytes) +
     ↪  _dataMemoryReservationStepInBytes;
119            }
120            if (minimumIndexReservedCapacity % _indexMemoryReservationStepInBytes > 0)
121            {
122                minimumIndexReservedCapacity = ((minimumIndexReservedCapacity /
     ↪  _indexMemoryReservationStepInBytes) * _indexMemoryReservationStepInBytes) +
     ↪  _indexMemoryReservationStepInBytes;
123            }
124            if (dataMemory.ReservedCapacity != minimumDataReservedCapacity)
125            {
126                dataMemory.ReservedCapacity = minimumDataReservedCapacity;
127            }
128            if (indexMemory.ReservedCapacity != minimumIndexReservedCapacity)
129            {
130                indexMemory.ReservedCapacity = minimumIndexReservedCapacity;
131            }
132            SetPointers(dataMemory, indexMemory);
133            header = ref GetHeaderReference();
134            // Ensure correctness _memory.UsedCapacity over _header->AllocatedLinks
135            // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
136            dataMemory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) *
     ↪  LinkDataPartSizeInBytes) + LinkDataPartSizeInBytes; // First link is read only
     ↪  zero link.
137            indexMemory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) *
     ↪  LinkIndexPartSizeInBytes) + LinkHeaderSizeInBytes;
138            // Ensure correctness _memory.ReservedLinks over _header->ReservedCapacity
139            // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
140            header.ReservedLinks = ConvertToAddress((dataMemory.ReservedCapacity -
     ↪  LinkDataPartSizeInBytes) / LinkDataPartSizeInBytes);
141        }
142
143        [MethodImpl(MethodImplOptions.AggressiveInlining)]
144        public virtual TLink Count(IList<TLink> restrictions)
145        {
146            // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
147            if (restrictions.Count == 0)
148            {
149                return Total;
150            }
151            var constants = Constants;
152            var any = constants.Any;
153            var index = restrictions[constants.IndexPart];
154            if (restrictions.Count == 1)
155            {
156                if (AreEqual(index, any))
157                {
158                    return Total;
159                }
160                return Exists(index) ? GetOne() : GetZero();
161            }
162            if (restrictions.Count == 2)
163            {
164                var value = restrictions[1];
165                if (AreEqual(index, any))
166                {
167                    if (AreEqual(value, any))
168                    {
169                        return Total; // Any - как отсутствие ограничения
170                    }
171                    var externalReferencesRange = constants.ExternalReferencesRange;
172                    if (externalReferencesRange.HasValue &&
     ↪  externalReferencesRange.Value.Contains(value))
173                    {
174                        return Add(ExternalSourcesTreeMethods.CountUsages(value),
     ↪  ExternalTargetsTreeMethods.CountUsages(value));
175                    }
176                    else
177                    {
178                        if (_useLinkedList)
179                        {
180                            return Add(InternalSourcesListMethods.CountUsages(value),
     ↪  InternalTargetsTreeMethods.CountUsages(value));
181                        }
182                        else
```

```
183                         {
184                             return Add(InternalSourcesTreeMethods.CountUsages(value),
                                  ↪   InternalTargetsTreeMethods.CountUsages(value));
185                         }
186                     }
187                 }
188                 else
189                 {
190                     if (!Exists(index))
191                     {
192                         return GetZero();
193                     }
194                     if (AreEqual(value, any))
195                     {
196                         return GetOne();
197                     }
198                     ref var storedLinkValue = ref GetLinkDataPartReference(index);
199                     if (AreEqual(storedLinkValue.Source, value) ||
                          ↪   AreEqual(storedLinkValue.Target, value))
200                     {
201                         return GetOne();
202                     }
203                     return GetZero();
204                 }
205             }
206             if (restrictions.Count == 3)
207             {
208                 var externalReferencesRange = constants.ExternalReferencesRange;
209                 var source = restrictions[constants.SourcePart];
210                 var target = restrictions[constants.TargetPart];
211                 if (AreEqual(index, any))
212                 {
213                     if (AreEqual(source, any) && AreEqual(target, any))
214                     {
215                         return Total;
216                     }
217                     else if (AreEqual(source, any))
218                     {
219                         if (externalReferencesRange.HasValue &&
                              ↪   externalReferencesRange.Value.Contains(target))
220                         {
221                             return ExternalTargetsTreeMethods.CountUsages(target);
222                         }
223                         else
224                         {
225                             return InternalTargetsTreeMethods.CountUsages(target);
226                         }
227                     }
228                     else if (AreEqual(target, any))
229                     {
230                         if (externalReferencesRange.HasValue &&
                              ↪   externalReferencesRange.Value.Contains(source))
231                         {
232                             return ExternalSourcesTreeMethods.CountUsages(source);
233                         }
234                         else
235                         {
236                             if (_useLinkedList)
237                             {
238                                 return InternalSourcesListMethods.CountUsages(source);
239                             }
240                             else
241                             {
242                                 return InternalSourcesTreeMethods.CountUsages(source);
243                             }
244                         }
245                     }
246                     else //if(source != Any && target != Any)
247                     {
248                         // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
249                         TLink link;
250                         if (externalReferencesRange.HasValue)
251                         {
252                             if (externalReferencesRange.Value.Contains(source) &&
                                  ↪   externalReferencesRange.Value.Contains(target))
253                             {
254                                 link = ExternalSourcesTreeMethods.Search(source, target);
255                             }
```

```
                    else if (externalReferencesRange.Value.Contains(source))
                    {
                        link = InternalTargetsTreeMethods.Search(source, target);
                    }
                    else if (externalReferencesRange.Value.Contains(target))
                    {
                        if (_useLinkedList)
                        {
                            link = ExternalSourcesTreeMethods.Search(source, target);
                        }
                        else
                        {
                            link = InternalSourcesTreeMethods.Search(source, target);
                        }
                    }
                    else
                    {
                        if (_useLinkedList ||
                        ↪   GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
                        ↪   InternalTargetsTreeMethods.CountUsages(target)))
                        {
                            link = InternalTargetsTreeMethods.Search(source, target);
                        }
                        else
                        {
                            link = InternalSourcesTreeMethods.Search(source, target);
                        }
                    }
                }
                else
                {
                    if (_useLinkedList ||
                    ↪   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();
```

```csharp
                    }
                    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
        {
            if (_useLinkedList)
            {
                return InternalSourcesListMethods.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))
            {
                if (_useLinkedList)
                {
                    link = ExternalSourcesTreeMethods.Search(source, target);
                }
                else
                {
                    link = InternalSourcesTreeMethods.Search(source, target);
                }
            }
            else
            {
                if (_useLinkedList ||
                  GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
                  InternalTargetsTreeMethods.CountUsages(target)))
                {
                    link = InternalTargetsTreeMethods.Search(source, target);
                }
                else
                {
                    link = InternalSourcesTreeMethods.Search(source, target);
                }
            }
        }
        else
```

```csharp
                        {
                            if (_useLinkedList ||
                            ↪  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))
            {
```

```csharp
                    if (externalReferencesRange.HasValue &&
                    ↪  externalReferencesRange.Value.Contains(source))
                    {
                        ExternalSourcesTreeMethods.Detach(ref rootAsSource, linkIndex);
                    }
                    else
                    {
                        if (_useLinkedList)
                        {
                            InternalSourcesListMethods.Detach(source, 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
                    {
                        if (_useLinkedList)
                        {
                            InternalSourcesListMethods.AttachAsLast(source, 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;
```

```csharp
618                    if (!AreEqual(freeLink, Constants.Null))
619                    {
620                        UnusedLinksListMethods.Detach(freeLink);
621                    }
622                    else
623                    {
624                        var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
625                        if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
626                        {
627                            throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
628                        }
629                        if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
630                        {
631                            _dataMemory.ReservedCapacity += _dataMemoryReservationStepInBytes;
632                            _indexMemory.ReservedCapacity += _indexMemoryReservationStepInBytes;
633                            SetPointers(_dataMemory, _indexMemory);
634                            header = ref GetHeaderReference();
635                            header.ReservedLinks = ConvertToAddress(_dataMemory.ReservedCapacity /
                                ↪ LinkDataPartSizeInBytes);
636                        }
637                        freeLink = header.AllocatedLinks = Increment(header.AllocatedLinks);
638                        _dataMemory.UsedCapacity += LinkDataPartSizeInBytes;
639                        _indexMemory.UsedCapacity += LinkIndexPartSizeInBytes;
640                    }
641                    return freeLink;
642                }
643
644                [MethodImpl(MethodImplOptions.AggressiveInlining)]
645                public virtual void Delete(IList<TLink> restrictions)
646                {
647                    ref var header = ref GetHeaderReference();
648                    var link = restrictions[Constants.IndexPart];
649                    if (LessThan(link, header.AllocatedLinks))
650                    {
651                        UnusedLinksListMethods.AttachAsFirst(link);
652                    }
653                    else if (AreEqual(link, header.AllocatedLinks))
654                    {
655                        header.AllocatedLinks = Decrement(header.AllocatedLinks);
656                        _dataMemory.UsedCapacity -= LinkDataPartSizeInBytes;
657                        _indexMemory.UsedCapacity -= LinkIndexPartSizeInBytes;
658                        // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
                            ↪ пока не дойдём до первой существующей связи
659                        // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
660                        while (GreaterThan(header.AllocatedLinks, GetZero()) &&
                            ↪ IsUnusedLink(header.AllocatedLinks))
661                        {
662                            UnusedLinksListMethods.Detach(header.AllocatedLinks);
663                            header.AllocatedLinks = Decrement(header.AllocatedLinks);
664                            _dataMemory.UsedCapacity -= LinkDataPartSizeInBytes;
665                            _indexMemory.UsedCapacity -= LinkIndexPartSizeInBytes;
666                        }
667                    }
668                }
669
670                [MethodImpl(MethodImplOptions.AggressiveInlining)]
671                public IList<TLink> GetLinkStruct(TLink linkIndex)
672                {
673                    ref var link = ref GetLinkDataPartReference(linkIndex);
674                    return new Link<TLink>(linkIndex, link.Source, link.Target);
675                }
676
677                /// <remarks>
678                /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
                    ↪ адрес реально поменялся
679                ///
680                /// Указатель this.links может быть в том же месте,
681                /// так как 0-я связь не используется и имеет такой же размер как Header,
682                /// поэтому header размещается в том же месте, что и 0-я связь
683                /// </remarks>
684                [MethodImpl(MethodImplOptions.AggressiveInlining)]
685                protected abstract void SetPointers(IResizableDirectMemory dataMemory,
                    ↪ IResizableDirectMemory indexMemory);
686
687                [MethodImpl(MethodImplOptions.AggressiveInlining)]
688                protected virtual void ResetPointers()
689                {
690                    InternalSourcesListMethods = null;
691                    InternalSourcesTreeMethods = null;
```

```csharp
                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.SizeAsTarget, 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);
```

```csharp
        [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
    }
}
```

## 1.46   ./csharp/Platform.Data.Doublets/Memory/Split/Generic/UnusedLinksListMethods.cs

```csharp
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> :
    ↪   AbsoluteCircularDoublyLinkedListMethods<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)]
```

```csharp
        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;
    }
}
```

## 1.47 ./csharp/Platform.Data.Doublets/Memory/Split/RawLinkDataPart.cs

```csharp
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);
    }
}
```

## 1.48 ./csharp/Platform.Data.Doublets/Memory/Split/RawLinkIndexPart.cs

```csharp
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>>
```

```
11          {
12              private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪   EqualityComparer<TLink>.Default;
13
14              public static readonly long SizeInBytes = Structure<RawLinkIndexPart<TLink>>.Size;
15
16              public TLink RootAsSource;
17              public TLink LeftAsSource;
18              public TLink RightAsSource;
19              public TLink SizeAsSource;
20              public TLink RootAsTarget;
21              public TLink LeftAsTarget;
22              public TLink RightAsTarget;
23              public TLink SizeAsTarget;
24
25              [MethodImpl(MethodImplOptions.AggressiveInlining)]
26              public override bool Equals(object obj) => obj is RawLinkIndexPart<TLink> link ?
        ↪   Equals(link) : false;
27
28              [MethodImpl(MethodImplOptions.AggressiveInlining)]
29              public bool Equals(RawLinkIndexPart<TLink> other)
30                  => _equalityComparer.Equals(RootAsSource, other.RootAsSource)
31                  && _equalityComparer.Equals(LeftAsSource, other.LeftAsSource)
32                  && _equalityComparer.Equals(RightAsSource, other.RightAsSource)
33                  && _equalityComparer.Equals(SizeAsSource, other.SizeAsSource)
34                  && _equalityComparer.Equals(RootAsTarget, other.RootAsTarget)
35                  && _equalityComparer.Equals(LeftAsTarget, other.LeftAsTarget)
36                  && _equalityComparer.Equals(RightAsTarget, other.RightAsTarget)
37                  && _equalityComparer.Equals(SizeAsTarget, other.SizeAsTarget);
38
39              [MethodImpl(MethodImplOptions.AggressiveInlining)]
40              public override int GetHashCode() => (RootAsSource, LeftAsSource, RightAsSource,
        ↪   SizeAsSource, RootAsTarget, LeftAsTarget, RightAsTarget, SizeAsTarget).GetHashCode();
41
42              [MethodImpl(MethodImplOptions.AggressiveInlining)]
43              public static bool operator ==(RawLinkIndexPart<TLink> left, RawLinkIndexPart<TLink>
        ↪   right) => left.Equals(right);
44
45              [MethodImpl(MethodImplOptions.AggressiveInlining)]
46              public static bool operator !=(RawLinkIndexPart<TLink> left, RawLinkIndexPart<TLink>
        ↪   right) => !(left == right);
47          }
48      }
```

## 1.49 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksRecursionlessSizeBalancedTree

```
1   using System.Runtime.CompilerServices;
2   using Platform.Data.Doublets.Memory.Split.Generic;
3   using TLink = System.UInt32;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.Memory.Split.Specific
8   {
9       public unsafe abstract class UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase :
        ↪   ExternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>, ILinksTreeMethods<TLink>
10      {
11          protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
12          protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
13          protected new readonly LinksHeader<TLink>* Header;
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          protected
        ↪   UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLink>
        ↪   constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
        ↪   linksIndexParts, LinksHeader<TLink>* header)
17              : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
18          {
19              LinksDataParts = linksDataParts;
20              LinksIndexParts = linksIndexParts;
21              Header = header;
22          }
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          protected override TLink GetZero() => 0U;
26
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          protected override bool EqualToZero(TLink value) => value == 0U;
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          protected override bool AreEqual(TLink first, TLink second) => first == second;
```

```csharp
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          protected override bool GreaterThanZero(TLink value) => value > 0U;
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          protected override bool GreaterThan(TLink first, TLink second) => first > second;
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          protected override bool GreaterOrEqualThan(TLink first, TLink second) => first >= second;
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          protected override bool GreaterOrEqualThanZero(TLink value) => true; // value >= 0 is
    ↪    always true for ulong
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected override bool LessOrEqualThanZero(TLink value) => value == 0UL; // value is
    ↪    always >= 0 for ulong
47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          protected override bool LessOrEqualThan(TLink first, TLink second) => first <= second;
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override bool LessThanZero(TLink value) => false; // value < 0 is always false
    ↪    for ulong
53
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          protected override bool LessThan(TLink first, TLink second) => first < second;
56
57          [MethodImpl(MethodImplOptions.AggressiveInlining)]
58          protected override TLink Increment(TLink value) => ++value;
59
60          [MethodImpl(MethodImplOptions.AggressiveInlining)]
61          protected override TLink Decrement(TLink value) => --value;
62
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          protected override TLink Add(TLink first, TLink second) => first + second;
65
66          [MethodImpl(MethodImplOptions.AggressiveInlining)]
67          protected override TLink Subtract(TLink first, TLink second) => first - second;
68
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          protected override ref LinksHeader<TLink> GetHeaderReference() => ref *Header;
71
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
    ↪    ref LinksDataParts[link];
74
75          [MethodImpl(MethodImplOptions.AggressiveInlining)]
76          protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
    ↪    ref LinksIndexParts[link];
77
78          [MethodImpl(MethodImplOptions.AggressiveInlining)]
79          protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
80          {
81              ref var firstLink = ref LinksDataParts[first];
82              ref var secondLink = ref LinksDataParts[second];
83              return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
    ↪    secondLink.Source, secondLink.Target);
84          }
85
86          [MethodImpl(MethodImplOptions.AggressiveInlining)]
87          protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
88          {
89              ref var firstLink = ref LinksDataParts[first];
90              ref var secondLink = ref LinksDataParts[second];
91              return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
    ↪    secondLink.Source, secondLink.Target);
92          }
93      }
94  }
```

## 1.50   ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksSizeBalancedTreeMethodsBase

```csharp
1   using System.Runtime.CompilerServices;
2   using Platform.Data.Doublets.Memory.Split.Generic;
3   using TLink = System.UInt32;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.Memory.Split.Specific
```

```csharp
    {
        public unsafe abstract class UInt32ExternalLinksSizeBalancedTreeMethodsBase :
            ExternalLinksSizeBalancedTreeMethodsBase<TLink>, ILinksTreeMethods<TLink>
        {
            protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
            protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
            protected new readonly LinksHeader<TLink>* Header;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected UInt32ExternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLink>
                constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
                linksIndexParts, LinksHeader<TLink>* header)
                : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
            {
                LinksDataParts = linksDataParts;
                LinksIndexParts = linksIndexParts;
                Header = header;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override TLink GetZero() => 0U;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
                ref LinksDataParts[link];

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
                ref LinksIndexParts[link];

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
```

```csharp
        {
            ref var firstLink = ref LinksDataParts[first];
            ref var secondLink = ref LinksDataParts[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 LinksDataParts[first];
            ref var secondLink = ref LinksDataParts[second];
            return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
            ↪  secondLink.Source, secondLink.Target);
        }
    }
}
```

1.51    ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksSourcesRecursionlessSizeBalanc

```csharp
using System.Runtime.CompilerServices;
using TLink = System.UInt32;

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe class UInt32ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods :
    ↪  UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public
        ↪  UInt32ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
        ↪  constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
        ↪  linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
        ↪  linksIndexParts, header) { }

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

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
48          protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
   ↪    TLink secondSource, TLink secondTarget)
49              => firstSource > secondSource || firstSource == secondSource && firstTarget >
   ↪    secondTarget;
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override void ClearNode(TLink node)
53          {
54              ref var link = ref LinksIndexParts[node];
55              link.LeftAsSource = Zero;
56              link.RightAsSource = Zero;
57              link.SizeAsSource = Zero;
58          }
59      }
60  }
```

## 1.52 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksSourcesSizeBalancedTreeMethe

```csharp
1  using System.Runtime.CompilerServices;
2  using TLink = System.UInt32;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.Split.Specific
7  {
8      public unsafe class UInt32ExternalLinksSourcesSizeBalancedTreeMethods :
   ↪    UInt32ExternalLinksSizeBalancedTreeMethodsBase
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public UInt32ExternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink>
   ↪    constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
   ↪    linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
   ↪    linksIndexParts, header) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected override ref TLink GetLeftReference(TLink node) => ref
   ↪    LinksIndexParts[node].LeftAsSource;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected override ref TLink GetRightReference(TLink node) => ref
   ↪    LinksIndexParts[node].RightAsSource;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsSource;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsSource;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override void SetLeft(TLink node, TLink left) =>
   ↪    LinksIndexParts[node].LeftAsSource = left;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void SetRight(TLink node, TLink right) =>
   ↪    LinksIndexParts[node].RightAsSource = right;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsSource;
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected override void SetSize(TLink node, TLink size) =>
   ↪    LinksIndexParts[node].SizeAsSource = size;
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected override TLink GetTreeRoot() => Header->RootAsSource;
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Source;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
   ↪    TLink secondSource, TLink secondTarget)
45             => firstSource < secondSource || firstSource == secondSource && firstTarget <
   ↪    secondTarget;
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
   ↪    TLink secondSource, TLink secondTarget)
49             => firstSource > secondSource || firstSource == secondSource && firstTarget >
   ↪    secondTarget;
```

```
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override void ClearNode(TLink node)
53          {
54              ref var link = ref LinksIndexParts[node];
55              link.LeftAsSource = Zero;
56              link.RightAsSource = Zero;
57              link.SizeAsSource = Zero;
58          }
59      }
60  }
```

## 1.53 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksTargetsRecursionlessSizeBalanc

```
1   using System.Runtime.CompilerServices;
2   using TLink = System.UInt32;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Memory.Split.Specific
7   {
8       public unsafe class UInt32ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods :
        ↪   UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase
9       {
10          [MethodImpl(MethodImplOptions.AggressiveInlining)]
11          public
            ↪   UInt32ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
            ↪   constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
            ↪   linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
            ↪   linksIndexParts, header) { }
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          protected override ref TLink GetLeftReference(TLink node) => ref
            ↪   LinksIndexParts[node].LeftAsTarget;
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          protected override ref TLink GetRightReference(TLink node) => ref
            ↪   LinksIndexParts[node].RightAsTarget;
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsTarget;
21
22          [MethodImpl(MethodImplOptions.AggressiveInlining)]
23          protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsTarget;
24
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          protected override void SetLeft(TLink node, TLink left) =>
            ↪   LinksIndexParts[node].LeftAsTarget = left;
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          protected override void SetRight(TLink node, TLink right) =>
            ↪   LinksIndexParts[node].RightAsTarget = right;
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsTarget;
33
34          [MethodImpl(MethodImplOptions.AggressiveInlining)]
35          protected override void SetSize(TLink node, TLink size) =>
            ↪   LinksIndexParts[node].SizeAsTarget = size;
36
37          [MethodImpl(MethodImplOptions.AggressiveInlining)]
38          protected override TLink GetTreeRoot() => Header->RootAsTarget;
39
40          [MethodImpl(MethodImplOptions.AggressiveInlining)]
41          protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Target;
42
43          [MethodImpl(MethodImplOptions.AggressiveInlining)]
44          protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
            ↪   TLink secondSource, TLink secondTarget)
45              => firstTarget < secondTarget || firstTarget == secondTarget && firstSource <
                ↪   secondSource;
46
47          [MethodImpl(MethodImplOptions.AggressiveInlining)]
48          protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
            ↪   TLink secondSource, TLink secondTarget)
49              => firstTarget > secondTarget || firstTarget == secondTarget && firstSource >
                ↪   secondSource;
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override void ClearNode(TLink node)
```

```
53              {
54                  ref var link = ref LinksIndexParts[node];
55                  link.LeftAsTarget = Zero;
56                  link.RightAsTarget = Zero;
57                  link.SizeAsTarget = Zero;
58              }
59          }
60      }
```

## 1.54 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksTargetsSizeBalancedTreeMetho

```
1   using System.Runtime.CompilerServices;
2   using TLink = System.UInt32;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Memory.Split.Specific
7   {
8       public unsafe class UInt32ExternalLinksTargetsSizeBalancedTreeMethods :
        ↪ UInt32ExternalLinksSizeBalancedTreeMethodsBase
9       {
10          [MethodImpl(MethodImplOptions.AggressiveInlining)]
11          public UInt32ExternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink>
           ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
           ↪ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
           ↪ linksIndexParts, header) { }
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          protected override ref TLink GetLeftReference(TLink node) => ref
           ↪ LinksIndexParts[node].LeftAsTarget;
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          protected override ref TLink GetRightReference(TLink node) => ref
           ↪ LinksIndexParts[node].RightAsTarget;
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsTarget;
21
22          [MethodImpl(MethodImplOptions.AggressiveInlining)]
23          protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsTarget;
24
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          protected override void SetLeft(TLink node, TLink left) =>
           ↪ LinksIndexParts[node].LeftAsTarget = left;
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          protected override void SetRight(TLink node, TLink right) =>
           ↪ LinksIndexParts[node].RightAsTarget = right;
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsTarget;
33
34          [MethodImpl(MethodImplOptions.AggressiveInlining)]
35          protected override void SetSize(TLink node, TLink size) =>
           ↪ LinksIndexParts[node].SizeAsTarget = size;
36
37          [MethodImpl(MethodImplOptions.AggressiveInlining)]
38          protected override TLink GetTreeRoot() => Header->RootAsTarget;
39
40          [MethodImpl(MethodImplOptions.AggressiveInlining)]
41          protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Target;
42
43          [MethodImpl(MethodImplOptions.AggressiveInlining)]
44          protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
           ↪ TLink secondSource, TLink secondTarget)
45              => firstTarget < secondTarget || firstTarget == secondTarget && firstSource <
               ↪ secondSource;
46
47          [MethodImpl(MethodImplOptions.AggressiveInlining)]
48          protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
           ↪ TLink secondSource, TLink secondTarget)
49              => firstTarget > secondTarget || firstTarget == secondTarget && firstSource >
               ↪ secondSource;
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override void ClearNode(TLink node)
53          {
54              ref var link = ref LinksIndexParts[node];
55              link.LeftAsTarget = Zero;
56              link.RightAsTarget = Zero;
```

```
57                  link.SizeAsTarget = Zero;
58              }
59          }
60      }
```

```
1   using System.Runtime.CompilerServices;
2   using Platform.Data.Doublets.Memory.Split.Generic;
3   using TLink = System.UInt32;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.Memory.Split.Specific
8   {
9       public unsafe abstract class UInt32InternalLinksRecursionlessSizeBalancedTreeMethodsBase :
     ↪   InternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
10      {
11          protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
12          protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
13          protected new readonly LinksHeader<TLink>* Header;
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          protected
     ↪   UInt32InternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLink>
     ↪   constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
     ↪   linksIndexParts, LinksHeader<TLink>* header)
17              : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
18          {
19              LinksDataParts = linksDataParts;
20              LinksIndexParts = linksIndexParts;
21              Header = header;
22          }
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          protected override TLink GetZero() => 0U;
26
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          protected override bool EqualToZero(TLink value) => value == 0U;
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          protected override bool AreEqual(TLink first, TLink second) => first == second;
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          protected override bool GreaterThanZero(TLink value) => value > 0U;
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          protected override bool GreaterThan(TLink first, TLink second) => first > second;
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          protected override bool GreaterOrEqualThan(TLink first, TLink second) => first >= second;
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          protected override bool GreaterOrEqualThanZero(TLink value) => true; // value >= 0 is
     ↪   always true for ulong
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected override bool LessOrEqualThanZero(TLink value) => value == 0UL; // value is
     ↪   always >= 0 for ulong
47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          protected override bool LessOrEqualThan(TLink first, TLink second) => first <= second;
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override bool LessThanZero(TLink value) => false; // value < 0 is always false
     ↪   for ulong
53
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          protected override bool LessThan(TLink first, TLink second) => first < second;
56
57          [MethodImpl(MethodImplOptions.AggressiveInlining)]
58          protected override TLink Increment(TLink value) => ++value;
59
60          [MethodImpl(MethodImplOptions.AggressiveInlining)]
61          protected override TLink Decrement(TLink value) => --value;
62
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          protected override TLink Add(TLink first, TLink second) => first + second;
65
66          [MethodImpl(MethodImplOptions.AggressiveInlining)]
67          protected override TLink Subtract(TLink first, TLink second) => first - second;
```

```
68
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
       ↪    ref LinksDataParts[link];
71
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
       ↪    ref LinksIndexParts[link];
74
75          [MethodImpl(MethodImplOptions.AggressiveInlining)]
76          protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second) =>
       ↪    GetKeyPartValue(first) < GetKeyPartValue(second);
77
78          [MethodImpl(MethodImplOptions.AggressiveInlining)]
79          protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second) =>
       ↪    GetKeyPartValue(first) > GetKeyPartValue(second);
80      }
81  }
```

## 1.56   ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksSizeBalancedTreeMethodsBase.

```
1   using System.Runtime.CompilerServices;
2   using Platform.Data.Doublets.Memory.Split.Generic;
3   using TLink = System.UInt32;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.Memory.Split.Specific
8   {
9       public unsafe abstract class UInt32InternalLinksSizeBalancedTreeMethodsBase :
       ↪    InternalLinksSizeBalancedTreeMethodsBase<TLink>
10      {
11          protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
12          protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
13          protected new readonly LinksHeader<TLink>* Header;
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          protected UInt32InternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLink>
       ↪    constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
       ↪    linksIndexParts, LinksHeader<TLink>* header)
17              : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
18          {
19              LinksDataParts = linksDataParts;
20              LinksIndexParts = linksIndexParts;
21              Header = header;
22          }
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          protected override TLink GetZero() => 0U;
26
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          protected override bool EqualToZero(TLink value) => value == 0U;
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          protected override bool AreEqual(TLink first, TLink second) => first == second;
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          protected override bool GreaterThanZero(TLink value) => value > 0U;
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          protected override bool GreaterThan(TLink first, TLink second) => first > second;
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          protected override bool GreaterOrEqualThan(TLink first, TLink second) => first >= second;
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          protected override bool GreaterOrEqualThanZero(TLink value) => true; // value >= 0 is
       ↪    always true for ulong
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected override bool LessOrEqualThanZero(TLink value) => value == 0UL; // value is
       ↪    always >= 0 for ulong
47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          protected override bool LessOrEqualThan(TLink first, TLink second) => first <= second;
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override bool LessThanZero(TLink value) => false; // value < 0 is always false
       ↪    for ulong
53
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
        protected override bool LessThan(TLink first, TLink second) => first < second;

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
            ref LinksDataParts[link];

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
            ref LinksIndexParts[link];

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

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

## 1.57 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksSourcesLinkedListMethods.cs

```csharp
using System.Runtime.CompilerServices;
using TLink = System.UInt32;

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

namespace Platform.Data.Doublets.Memory.Split.Generic
{
    public unsafe class UInt32InternalLinksSourcesLinkedListMethods :
        InternalLinksSourcesLinkedListMethods<TLink>
    {
        private readonly RawLinkDataPart<TLink>* _linksDataParts;
        private readonly RawLinkIndexPart<TLink>* _linksIndexParts;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32InternalLinksSourcesLinkedListMethods(LinksConstants<TLink> constants,
            RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>* linksIndexParts)
            : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts)
        {
            _linksDataParts = linksDataParts;
            _linksIndexParts = linksIndexParts;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
            ref _linksDataParts[link];

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
            ref _linksIndexParts[link];
    }
}
```

## 1.58 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksSourcesRecursionlessSizeBalanc

```csharp
using System.Runtime.CompilerServices;
using TLink = System.UInt32;

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe class UInt32InternalLinksSourcesRecursionlessSizeBalancedTreeMethods :
        UInt32InternalLinksRecursionlessSizeBalancedTreeMethodsBase
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
        public
    ↪   UInt32InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
    ↪   constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
    ↪   linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
    ↪   linksIndexParts, header) { }

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetKeyPartValue(TLink node) => LinksDataParts[node].Target;

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

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

```csharp
using System.Runtime.CompilerServices;
using TLink = System.UInt32;

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe class UInt32InternalLinksSourcesSizeBalancedTreeMethods :
    ↪   UInt32InternalLinksSizeBalancedTreeMethodsBase
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32InternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink>
    ↪   constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
    ↪   linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
    ↪   linksIndexParts, header) { }

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetKeyPartValue(TLink node) => LinksDataParts[node].Target;

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

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

## 1.60   ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksTargetsRecursionlessSizeBalanc

```csharp
using System.Runtime.CompilerServices;
using TLink = System.UInt32;

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe class UInt32InternalLinksTargetsRecursionlessSizeBalancedTreeMethods :
    ↪    UInt32InternalLinksRecursionlessSizeBalancedTreeMethodsBase
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public
        ↪    UInt32InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
        ↪    constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
        ↪    linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
        ↪    linksIndexParts, header) { }

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

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

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

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

```
24
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          protected override void SetLeft(TLink node, TLink left) =>
    ↪       LinksIndexParts[node].LeftAsTarget = left;
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          protected override void SetRight(TLink node, TLink right) =>
    ↪       LinksIndexParts[node].RightAsTarget = right;
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsTarget;
33
34          [MethodImpl(MethodImplOptions.AggressiveInlining)]
35          protected override void SetSize(TLink node, TLink size) =>
    ↪       LinksIndexParts[node].SizeAsTarget = size;
36
37          [MethodImpl(MethodImplOptions.AggressiveInlining)]
38          protected override TLink GetTreeRoot(TLink node) => LinksIndexParts[node].RootAsTarget;
39
40          [MethodImpl(MethodImplOptions.AggressiveInlining)]
41          protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Target;
42
43          [MethodImpl(MethodImplOptions.AggressiveInlining)]
44          protected override TLink GetKeyPartValue(TLink node) => LinksDataParts[node].Source;
45
46          [MethodImpl(MethodImplOptions.AggressiveInlining)]
47          protected override void ClearNode(TLink node)
48          {
49              ref var link = ref LinksIndexParts[node];
50              link.LeftAsTarget = Zero;
51              link.RightAsTarget = Zero;
52              link.SizeAsTarget = Zero;
53          }
54
55          public override TLink Search(TLink source, TLink target) =>
    ↪       SearchCore(GetTreeRoot(target), source);
56      }
57  }
```

## 1.61 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksTargetsSizeBalancedTreeMetho

```
1   using System.Runtime.CompilerServices;
2   using TLink = System.UInt32;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Memory.Split.Specific
7   {
8       public unsafe class UInt32InternalLinksTargetsSizeBalancedTreeMethods :
    ↪       UInt32InternalLinksSizeBalancedTreeMethodsBase
9       {
10          [MethodImpl(MethodImplOptions.AggressiveInlining)]
11          public UInt32InternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink>
    ↪       constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
    ↪       linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
    ↪       linksIndexParts, header) { }
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          protected override ref TLink GetLeftReference(TLink node) => ref
    ↪       LinksIndexParts[node].LeftAsTarget;
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          protected override ref TLink GetRightReference(TLink node) => ref
    ↪       LinksIndexParts[node].RightAsTarget;
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsTarget;
21
22          [MethodImpl(MethodImplOptions.AggressiveInlining)]
23          protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsTarget;
24
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          protected override void SetLeft(TLink node, TLink left) =>
    ↪       LinksIndexParts[node].LeftAsTarget = left;
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          protected override void SetRight(TLink node, TLink right) =>
    ↪       LinksIndexParts[node].RightAsTarget = right;
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
        protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsTarget;

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetKeyPartValue(TLink node) => LinksDataParts[node].Source;

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

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

## 1.62 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32SplitMemoryLinks.cs

```csharp
using System;
using System.Runtime.CompilerServices;
using Platform.Singletons;
using Platform.Memory;
using Platform.Data.Doublets.Memory.Split.Generic;
using TLink = System.UInt32;

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe class UInt32SplitMemoryLinks : 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 LinksHeader<TLink>* _header;
        private RawLinkDataPart<TLink>* _linksDataParts;
        private RawLinkIndexPart<TLink>* _linksIndexParts;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
        ↪   indexMemory) : this(dataMemory, indexMemory, DefaultLinksSizeStep) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
        ↪   indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
        ↪   memoryReservationStep, Default<LinksConstants<TLink>>.Instance,
        ↪   IndexTreeType.Default, useLinkedList: true) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
        ↪   indexMemory, long memoryReservationStep, LinksConstants<TLink> constants) :
        ↪   this(dataMemory, indexMemory, memoryReservationStep, constants,
        ↪   IndexTreeType.Default, useLinkedList: true) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
        ↪   indexMemory, long memoryReservationStep, LinksConstants<TLink> constants,
        ↪   IndexTreeType indexTreeType, bool useLinkedList) : base(dataMemory, indexMemory,
        ↪   memoryReservationStep, constants, useLinkedList)
        {
            if (indexTreeType == IndexTreeType.SizeBalancedTree)
            {
                _createInternalSourceTreeMethods = () => new
                ↪   UInt32InternalLinksSourcesSizeBalancedTreeMethods(Constants,
                ↪   _linksDataParts, _linksIndexParts, _header);
```

```csharp
                _createExternalSourceTreeMethods = () => new
                ↪  UInt32ExternalLinksSourcesSizeBalancedTreeMethods(Constants,
                ↪  _linksDataParts, _linksIndexParts, _header);
                _createInternalTargetTreeMethods = () => new
                ↪  UInt32InternalLinksTargetsSizeBalancedTreeMethods(Constants,
                ↪  _linksDataParts, _linksIndexParts, _header);
                _createExternalTargetTreeMethods = () => new
                ↪  UInt32ExternalLinksTargetsSizeBalancedTreeMethods(Constants,
                ↪  _linksDataParts, _linksIndexParts, _header);
            }
            else
            {
                _createInternalSourceTreeMethods = () => new
                ↪  UInt32InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(Constants,
                ↪  _linksDataParts, _linksIndexParts, _header);
                _createExternalSourceTreeMethods = () => new
                ↪  UInt32ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(Constants,
                ↪  _linksDataParts, _linksIndexParts, _header);
                _createInternalTargetTreeMethods = () => new
                ↪  UInt32InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(Constants,
                ↪  _linksDataParts, _linksIndexParts, _header);
                _createExternalTargetTreeMethods = () => new
                ↪  UInt32ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(Constants,
                ↪  _linksDataParts, _linksIndexParts, _header);
            }
            Init(dataMemory, indexMemory);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetPointers(IResizableDirectMemory dataMemory,
        ↪  IResizableDirectMemory indexMemory)
        {
            _linksDataParts = (RawLinkDataPart<TLink>*)dataMemory.Pointer;
            _linksIndexParts = (RawLinkIndexPart<TLink>*)indexMemory.Pointer;
            _header = (LinksHeader<TLink>*)indexMemory.Pointer;
            if (_useLinkedList)
            {
                InternalSourcesListMethods = new
                ↪  UInt32InternalLinksSourcesLinkedListMethods(Constants, _linksDataParts,
                ↪  _linksIndexParts);
            }
            else
            {
                InternalSourcesTreeMethods = _createInternalSourceTreeMethods();
            }
            ExternalSourcesTreeMethods = _createExternalSourceTreeMethods();
            InternalTargetsTreeMethods = _createInternalTargetTreeMethods();
            ExternalTargetsTreeMethods = _createExternalTargetTreeMethods();
            UnusedLinksListMethods = new UInt32UnusedLinksListMethods(_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 *_header;

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
        ↪  linkIndex) => ref _linksIndexParts[linkIndex];

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
 96        protected override bool LessOrEqualThan(TLink first, TLink second) => first <= second;
 97
 98        [MethodImpl(MethodImplOptions.AggressiveInlining)]
 99        protected override bool GreaterThan(TLink first, TLink second) => first > second;
100
101        [MethodImpl(MethodImplOptions.AggressiveInlining)]
102        protected override bool GreaterOrEqualThan(TLink first, TLink second) => first >= second;
103
104        [MethodImpl(MethodImplOptions.AggressiveInlining)]
105        protected override TLink GetZero() => 0U;
106
107        [MethodImpl(MethodImplOptions.AggressiveInlining)]
108        protected override TLink GetOne() => 1U;
109
110        [MethodImpl(MethodImplOptions.AggressiveInlining)]
111        protected override long ConvertToInt64(TLink value) => value;
112
113        [MethodImpl(MethodImplOptions.AggressiveInlining)]
114        protected override TLink ConvertToAddress(long value) => (TLink)value;
115
116        [MethodImpl(MethodImplOptions.AggressiveInlining)]
117        protected override TLink Add(TLink first, TLink second) => first + second;
118
119        [MethodImpl(MethodImplOptions.AggressiveInlining)]
120        protected override TLink Subtract(TLink first, TLink second) => first - second;
121
122        [MethodImpl(MethodImplOptions.AggressiveInlining)]
123        protected override TLink Increment(TLink link) => ++link;
124
125        [MethodImpl(MethodImplOptions.AggressiveInlining)]
126        protected override TLink Decrement(TLink link) => --link;
127    }
128 }
```

## 1.63  ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32UnusedLinksListMethods.cs

```csharp
 1 using System.Runtime.CompilerServices;
 2 using Platform.Data.Doublets.Memory.Split.Generic;
 3 using TLink = System.UInt32;
 4
 5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
 6
 7 namespace Platform.Data.Doublets.Memory.Split.Specific
 8 {
 9    public unsafe class UInt32UnusedLinksListMethods : UnusedLinksListMethods<TLink>
10    {
11        private readonly RawLinkDataPart<TLink>* _links;
12        private readonly LinksHeader<TLink>* _header;
13
14        [MethodImpl(MethodImplOptions.AggressiveInlining)]
15        public UInt32UnusedLinksListMethods(RawLinkDataPart<TLink>* links, LinksHeader<TLink>*
           ↪  header)
16            : base((byte*)links, (byte*)header)
17        {
18            _links = links;
19            _header = header;
20        }
21
22        [MethodImpl(MethodImplOptions.AggressiveInlining)]
23        protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
           ↪  ref _links[link];
24
25        [MethodImpl(MethodImplOptions.AggressiveInlining)]
26        protected override ref LinksHeader<TLink> GetHeaderReference() => ref *_header;
27    }
28 }
```

## 1.64  ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksRecursionlessSizeBalancedTree

```csharp
 1 using System.Runtime.CompilerServices;
 2 using Platform.Data.Doublets.Memory.Split.Generic;
 3 using TLink = System.UInt64;
 4
 5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
 6
 7 namespace Platform.Data.Doublets.Memory.Split.Specific
 8 {
 9    public unsafe abstract class UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase :
       ↪  ExternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>, ILinksTreeMethods<TLink>
10    {
11        protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
12        protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
```

```csharp
        protected new readonly LinksHeader<TLink>* Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected
        ↪  UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLink>
        ↪  constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
        ↪  linksIndexParts, LinksHeader<TLink>* header)
            : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
        {
            LinksDataParts = linksDataParts;
            LinksIndexParts = linksIndexParts;
            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 ref LinksHeader<TLink> GetHeaderReference() => ref *Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
        ↪   ref LinksDataParts[link];

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
        ↪   ref LinksIndexParts[link];

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

```csharp
        }

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

**1.65** ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksSizeBalancedTreeMethodsBase

```csharp
using System.Runtime.CompilerServices;
using Platform.Data.Doublets.Memory.Split.Generic;
using TLink = System.UInt64;

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe abstract class UInt64ExternalLinksSizeBalancedTreeMethodsBase :
    ↪  ExternalLinksSizeBalancedTreeMethodsBase<TLink>, ILinksTreeMethods<TLink>
    {
        protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
        protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
        protected new readonly LinksHeader<TLink>* Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected UInt64ExternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLink>
        ↪  constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
        ↪  linksIndexParts, LinksHeader<TLink>* header)
            : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
        {
            LinksDataParts = linksDataParts;
            LinksIndexParts = linksIndexParts;
            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)]
```

```csharp
        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 ref LinksHeader<TLink> GetHeaderReference() => ref *Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
            ref LinksDataParts[link];

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
            ref LinksIndexParts[link];

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

## 1.66 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksSourcesRecursionlessSizeBalanc

```csharp
using System.Runtime.CompilerServices;
using TLink = System.UInt64;

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe class UInt64ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods :
        UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public
            UInt64ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
            constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
            linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
            linksIndexParts, header) { }

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

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

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

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

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

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

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

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

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

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

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

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

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

## 1.67 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksSourcesSizeBalancedTreeMetho

```csharp
using System.Runtime.CompilerServices;
using TLink = System.UInt64;

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe class UInt64ExternalLinksSourcesSizeBalancedTreeMethods :
        UInt64ExternalLinksSizeBalancedTreeMethodsBase
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64ExternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink>
            constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
            linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
            linksIndexParts, header) { }

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
35      protected override void SetSize(TLink node, TLink size) =>
        ↪ LinksIndexParts[node].SizeAsSource = size;
36
37      [MethodImpl(MethodImplOptions.AggressiveInlining)]
38      protected override TLink GetTreeRoot() => Header->RootAsSource;
39
40      [MethodImpl(MethodImplOptions.AggressiveInlining)]
41      protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Source;
42
43      [MethodImpl(MethodImplOptions.AggressiveInlining)]
44      protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
        ↪ TLink secondSource, TLink secondTarget)
45          => firstSource < secondSource || firstSource == secondSource && firstTarget <
            ↪ secondTarget;
46
47      [MethodImpl(MethodImplOptions.AggressiveInlining)]
48      protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
        ↪ TLink secondSource, TLink secondTarget)
49          => firstSource > secondSource || firstSource == secondSource && firstTarget >
            ↪ secondTarget;
50
51      [MethodImpl(MethodImplOptions.AggressiveInlining)]
52      protected override void ClearNode(TLink node)
53      {
54          ref var link = ref LinksIndexParts[node];
55          link.LeftAsSource = Zero;
56          link.RightAsSource = Zero;
57          link.SizeAsSource = Zero;
58      }
59  }
60 }
```

## 1.68 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksTargetsRecursionlessSizeBalan

```
1  using System.Runtime.CompilerServices;
2  using TLink = System.UInt64;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.Split.Specific
7  {
8      public unsafe class UInt64ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods :
        ↪ UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public
           ↪ UInt64ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
           ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
           ↪ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
           ↪ linksIndexParts, header) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected override ref TLink GetLeftReference(TLink node) => ref
           ↪ LinksIndexParts[node].LeftAsTarget;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected override ref TLink GetRightReference(TLink node) => ref
           ↪ LinksIndexParts[node].RightAsTarget;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsTarget;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsTarget;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override void SetLeft(TLink node, TLink left) =>
           ↪ LinksIndexParts[node].LeftAsTarget = left;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void SetRight(TLink node, TLink right) =>
           ↪ LinksIndexParts[node].RightAsTarget = right;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsTarget;
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected override void SetSize(TLink node, TLink size) =>
           ↪ LinksIndexParts[node].SizeAsTarget = size;
36
```

```
37        [MethodImpl(MethodImplOptions.AggressiveInlining)]
38        protected override TLink GetTreeRoot() => Header->RootAsTarget;
39
40        [MethodImpl(MethodImplOptions.AggressiveInlining)]
41        protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Target;
42
43        [MethodImpl(MethodImplOptions.AggressiveInlining)]
44        protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
       ↪  TLink secondSource, TLink secondTarget)
45            => firstTarget < secondTarget || firstTarget == secondTarget && firstSource <
              ↪  secondSource;
46
47        [MethodImpl(MethodImplOptions.AggressiveInlining)]
48        protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
       ↪  TLink secondSource, TLink secondTarget)
49            => firstTarget > secondTarget || firstTarget == secondTarget && firstSource >
              ↪  secondSource;
50
51        [MethodImpl(MethodImplOptions.AggressiveInlining)]
52        protected override void ClearNode(TLink node)
53        {
54            ref var link = ref LinksIndexParts[node];
55            link.LeftAsTarget = Zero;
56            link.RightAsTarget = Zero;
57            link.SizeAsTarget = Zero;
58        }
59    }
60 }
```

## 1.69 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksTargetsSizeBalancedTreeMetho

```
1  using System.Runtime.CompilerServices;
2  using TLink = System.UInt64;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.Split.Specific
7  {
8      public unsafe class UInt64ExternalLinksTargetsSizeBalancedTreeMethods :
       ↪  UInt64ExternalLinksSizeBalancedTreeMethodsBase
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public UInt64ExternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink>
       ↪  constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
       ↪  linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
       ↪  linksIndexParts, header) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected override ref TLink GetLeftReference(TLink node) => ref
       ↪  LinksIndexParts[node].LeftAsTarget;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected override ref TLink GetRightReference(TLink node) => ref
       ↪  LinksIndexParts[node].RightAsTarget;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsTarget;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsTarget;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override void SetLeft(TLink node, TLink left) =>
       ↪  LinksIndexParts[node].LeftAsTarget = left;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void SetRight(TLink node, TLink right) =>
       ↪  LinksIndexParts[node].RightAsTarget = right;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsTarget;
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected override void SetSize(TLink node, TLink size) =>
       ↪  LinksIndexParts[node].SizeAsTarget = size;
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected override TLink GetTreeRoot() => Header->RootAsTarget;
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
        protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Target;

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

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

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

## 1.70    ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksRecursionlessSizeBalancedTreeN

```csharp
using System.Runtime.CompilerServices;
using Platform.Data.Doublets.Memory.Split.Generic;
using TLink = System.UInt64;

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe abstract class UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase :
    ↪ InternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
    {
        protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
        protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
        protected new readonly LinksHeader<TLink>* Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected
        ↪ UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLink>
        ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
        ↪ linksIndexParts, LinksHeader<TLink>* header)
            : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
        {
            LinksDataParts = linksDataParts;
            LinksIndexParts = linksIndexParts;
            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)]
```

```
49        protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
50
51        [MethodImpl(MethodImplOptions.AggressiveInlining)]
52        protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
   ↪   for ulong
53
54        [MethodImpl(MethodImplOptions.AggressiveInlining)]
55        protected override bool LessThan(ulong first, ulong second) => first < second;
56
57        [MethodImpl(MethodImplOptions.AggressiveInlining)]
58        protected override ulong Increment(ulong value) => ++value;
59
60        [MethodImpl(MethodImplOptions.AggressiveInlining)]
61        protected override ulong Decrement(ulong value) => --value;
62
63        [MethodImpl(MethodImplOptions.AggressiveInlining)]
64        protected override ulong Add(ulong first, ulong second) => first + second;
65
66        [MethodImpl(MethodImplOptions.AggressiveInlining)]
67        protected override ulong Subtract(ulong first, ulong second) => first - second;
68
69        [MethodImpl(MethodImplOptions.AggressiveInlining)]
70        protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
   ↪   ref LinksDataParts[link];
71
72        [MethodImpl(MethodImplOptions.AggressiveInlining)]
73        protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
   ↪   ref LinksIndexParts[link];
74
75        [MethodImpl(MethodImplOptions.AggressiveInlining)]
76        protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second) =>
   ↪   GetKeyPartValue(first) < GetKeyPartValue(second);
77
78        [MethodImpl(MethodImplOptions.AggressiveInlining)]
79        protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second) =>
   ↪   GetKeyPartValue(first) > GetKeyPartValue(second);
80    }
81  }
```

## 1.71  ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksSizeBalancedTreeMethodsBase.

```
1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.Memory.Split.Generic;
3  using TLink = System.UInt64;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Memory.Split.Specific
8  {
9      public unsafe abstract class UInt64InternalLinksSizeBalancedTreeMethodsBase :
   ↪   InternalLinksSizeBalancedTreeMethodsBase<TLink>
10     {
11         protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
12         protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
13         protected new readonly LinksHeader<TLink>* Header;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         protected UInt64InternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLink>
   ↪   constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
   ↪   linksIndexParts, LinksHeader<TLink>* header)
17             : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
18         {
19             LinksDataParts = linksDataParts;
20             LinksIndexParts = linksIndexParts;
21             Header = header;
22         }
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override ulong GetZero() => 0UL;
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override bool EqualToZero(ulong value) => value == 0UL;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected override bool AreEqual(ulong first, ulong second) => first == second;
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected override bool GreaterThanZero(ulong value) => value > 0UL;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override bool GreaterThan(ulong first, ulong second) => first > second;
```

```csharp
        [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 ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
            ref LinksDataParts[link];

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
            ref LinksIndexParts[link];

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

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

## 1.72   ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksSourcesLinkedListMethods.cs

```csharp
using System.Runtime.CompilerServices;
using TLink = System.UInt64;

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

namespace Platform.Data.Doublets.Memory.Split.Generic
{
    public unsafe class UInt64InternalLinksSourcesLinkedListMethods :
        InternalLinksSourcesLinkedListMethods<TLink>
    {
        private readonly RawLinkDataPart<TLink>* _linksDataParts;
        private readonly RawLinkIndexPart<TLink>* _linksIndexParts;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64InternalLinksSourcesLinkedListMethods(LinksConstants<TLink> constants,
            RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>* linksIndexParts)
            : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts)
        {
            _linksDataParts = linksDataParts;
            _linksIndexParts = linksIndexParts;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
            ref _linksDataParts[link];
```

```
24        [MethodImpl(MethodImplOptions.AggressiveInlining)]
25        protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
   ↪    ref _linksIndexParts[link];
26    }
27  }
```

## 1.73 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksSourcesRecursionlessSizeBalanc

```
1  using System.Runtime.CompilerServices;
2  using TLink = System.UInt64;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.Split.Specific
7  {
8      public unsafe class UInt64InternalLinksSourcesRecursionlessSizeBalancedTreeMethods :
   ↪    UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase
9      {
10          [MethodImpl(MethodImplOptions.AggressiveInlining)]
11          public
   ↪    UInt64InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
   ↪    constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
   ↪    linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
   ↪    linksIndexParts, header) { }
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          protected override ref TLink GetLeftReference(TLink node) => ref
   ↪    LinksIndexParts[node].LeftAsSource;
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          protected override ref TLink GetRightReference(TLink node) => ref
   ↪    LinksIndexParts[node].RightAsSource;
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsSource;
21
22          [MethodImpl(MethodImplOptions.AggressiveInlining)]
23          protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsSource;
24
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          protected override void SetLeft(TLink node, TLink left) =>
   ↪    LinksIndexParts[node].LeftAsSource = left;
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          protected override void SetRight(TLink node, TLink right) =>
   ↪    LinksIndexParts[node].RightAsSource = right;
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsSource;
33
34          [MethodImpl(MethodImplOptions.AggressiveInlining)]
35          protected override void SetSize(TLink node, TLink size) =>
   ↪    LinksIndexParts[node].SizeAsSource = size;
36
37          [MethodImpl(MethodImplOptions.AggressiveInlining)]
38          protected override TLink GetTreeRoot(TLink node) => LinksIndexParts[node].RootAsSource;
39
40          [MethodImpl(MethodImplOptions.AggressiveInlining)]
41          protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Source;
42
43          [MethodImpl(MethodImplOptions.AggressiveInlining)]
44          protected override TLink GetKeyPartValue(TLink node) => LinksDataParts[node].Target;
45
46          [MethodImpl(MethodImplOptions.AggressiveInlining)]
47          protected override void ClearNode(TLink node)
48          {
49              ref var link = ref LinksIndexParts[node];
50              link.LeftAsSource = Zero;
51              link.RightAsSource = Zero;
52              link.SizeAsSource = Zero;
53          }
54
55          public override TLink Search(TLink source, TLink target) =>
   ↪    SearchCore(GetTreeRoot(source), target);
56      }
57  }
```

## 1.74 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksSourcesSizeBalancedTreeMetho

```
1  using System.Runtime.CompilerServices;
2  using TLink = System.UInt64;
```

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe class UInt64InternalLinksSourcesSizeBalancedTreeMethods :
        UInt64InternalLinksSizeBalancedTreeMethodsBase
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64InternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink>
            constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
            linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
            linksIndexParts, header) { }

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetKeyPartValue(TLink node) => LinksDataParts[node].Target;

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

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

## 1.75  ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksTargetsRecursionlessSizeBalanc

```csharp
using System.Runtime.CompilerServices;
using TLink = System.UInt64;

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe class UInt64InternalLinksTargetsRecursionlessSizeBalancedTreeMethods :
        UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
            public
            ↪   UInt64InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
            ↪   constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
            ↪   linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
            ↪   linksIndexParts, header) { }

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

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

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

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

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

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

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

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

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

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override TLink GetKeyPartValue(TLink node) => LinksDataParts[node].Source;

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

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

## 1.76 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksTargetsSizeBalancedTreeMetho

```csharp
using System.Runtime.CompilerServices;
using TLink = System.UInt64;

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe class UInt64InternalLinksTargetsSizeBalancedTreeMethods :
    ↪   UInt64InternalLinksSizeBalancedTreeMethodsBase
    {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public UInt64InternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink>
            ↪   constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
            ↪   linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
            ↪   linksIndexParts, header) { }

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetKeyPartValue(TLink node) => LinksDataParts[node].Source;

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

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

## 1.77  ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64SplitMemoryLinks.cs

```csharp
using System;
using System.Runtime.CompilerServices;
using Platform.Singletons;
using Platform.Memory;
using Platform.Data.Doublets.Memory.Split.Generic;
using TLink = System.UInt64;

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe class UInt64SplitMemoryLinks : 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 LinksHeader<ulong>* _header;
        private RawLinkDataPart<ulong>* _linksDataParts;
        private RawLinkIndexPart<ulong>* _linksIndexParts;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
        ↪   indexMemory) : this(dataMemory, indexMemory, DefaultLinksSizeStep) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
        ↪   indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
        ↪   memoryReservationStep, Default<LinksConstants<TLink>>.Instance,
        ↪   IndexTreeType.Default, useLinkedList: true) { }

```

```csharp
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          public UInt64SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
   ↪ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants) :
   ↪ this(dataMemory, indexMemory, memoryReservationStep, constants,
   ↪ IndexTreeType.Default, useLinkedList: true) { }

31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          public UInt64SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
   ↪ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants,
   ↪ IndexTreeType indexTreeType, bool useLinkedList) : base(dataMemory, indexMemory,
   ↪ memoryReservationStep, constants, useLinkedList)
33          {
34              if (indexTreeType == IndexTreeType.SizeBalancedTree)
35              {
36                  _createInternalSourceTreeMethods = () => new
                     ↪ UInt64InternalLinksSourcesSizeBalancedTreeMethods(Constants,
                     ↪ _linksDataParts, _linksIndexParts, _header);
37                  _createExternalSourceTreeMethods = () => new
                     ↪ UInt64ExternalLinksSourcesSizeBalancedTreeMethods(Constants,
                     ↪ _linksDataParts, _linksIndexParts, _header);
38                  _createInternalTargetTreeMethods = () => new
                     ↪ UInt64InternalLinksTargetsSizeBalancedTreeMethods(Constants,
                     ↪ _linksDataParts, _linksIndexParts, _header);
39                  _createExternalTargetTreeMethods = () => new
                     ↪ UInt64ExternalLinksTargetsSizeBalancedTreeMethods(Constants,
                     ↪ _linksDataParts, _linksIndexParts, _header);
40              }
41              else
42              {
43                  _createInternalSourceTreeMethods = () => new
                     ↪ UInt64InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(Constants,
                     ↪ _linksDataParts, _linksIndexParts, _header);
44                  _createExternalSourceTreeMethods = () => new
                     ↪ UInt64ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(Constants,
                     ↪ _linksDataParts, _linksIndexParts, _header);
45                  _createInternalTargetTreeMethods = () => new
                     ↪ UInt64InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(Constants,
                     ↪ _linksDataParts, _linksIndexParts, _header);
46                  _createExternalTargetTreeMethods = () => new
                     ↪ UInt64ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(Constants,
                     ↪ _linksDataParts, _linksIndexParts, _header);
47              }
48              Init(dataMemory, indexMemory);
49          }

51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override void SetPointers(IResizableDirectMemory dataMemory,
   ↪ IResizableDirectMemory indexMemory)
53          {
54              _linksDataParts = (RawLinkDataPart<TLink>*)dataMemory.Pointer;
55              _linksIndexParts = (RawLinkIndexPart<TLink>*)indexMemory.Pointer;
56              _header = (LinksHeader<TLink>*)indexMemory.Pointer;
57              if (_useLinkedList)
58              {
59                  InternalSourcesListMethods = new
                     ↪ UInt64InternalLinksSourcesLinkedListMethods(Constants, _linksDataParts,
                     ↪ _linksIndexParts);
60              }
61              else
62              {
63                  InternalSourcesTreeMethods = _createInternalSourceTreeMethods();
64              }
65              ExternalSourcesTreeMethods = _createExternalSourceTreeMethods();
66              InternalTargetsTreeMethods = _createInternalTargetTreeMethods();
67              ExternalTargetsTreeMethods = _createExternalTargetTreeMethods();
68              UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_linksDataParts, _header);
69          }

71          [MethodImpl(MethodImplOptions.AggressiveInlining)]
72          protected override void ResetPointers()
73          {
74              base.ResetPointers();
75              _linksDataParts = null;
76              _linksIndexParts = null;
77              _header = null;
78          }
79
```

```csharp
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref LinksHeader<TLink> GetHeaderReference() => ref *_header;

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

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

## 1.78  ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64UnusedLinksListMethods.cs

```csharp
using System.Runtime.CompilerServices;
using Platform.Data.Doublets.Memory.Split.Generic;
using TLink = System.UInt64;

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe class UInt64UnusedLinksListMethods : UnusedLinksListMethods<TLink>
    {
        private readonly RawLinkDataPart<ulong>* _links;
        private readonly LinksHeader<ulong>* _header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64UnusedLinksListMethods(RawLinkDataPart<ulong>* links, LinksHeader<ulong>*
        ↪   header)
            : base((byte*)links, (byte*)header)
        {
            _links = links;
            _header = header;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
        ↪   ref _links[link];

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
26          protected override ref LinksHeader<TLink> GetHeaderReference() => ref *_header;
27      }
28  }
```

## 1.79   ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksAvlBalancedTreeMethodsBase.cs

```csharp
1   using System;
2   using System.Text;
3   using System.Collections.Generic;
4   using System.Runtime.CompilerServices;
5   using Platform.Collections.Methods.Trees;
6   using Platform.Converters;
7   using Platform.Numbers;
8   using static System.Runtime.CompilerServices.Unsafe;
9
10  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12  namespace Platform.Data.Doublets.Memory.United.Generic
13  {
14      public unsafe abstract class LinksAvlBalancedTreeMethodsBase<TLink> :
        ↪  SizedAndThreadedAVLBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
15      {
16          private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
            ↪  UncheckedConverter<TLink, long>.Default;
17          private static readonly UncheckedConverter<TLink, int> _addressToInt32Converter =
            ↪  UncheckedConverter<TLink, int>.Default;
18          private static readonly UncheckedConverter<bool, TLink> _boolToAddressConverter =
            ↪  UncheckedConverter<bool, TLink>.Default;
19          private static readonly UncheckedConverter<TLink, bool> _addressToBoolConverter =
            ↪  UncheckedConverter<TLink, bool>.Default;
20          private static readonly UncheckedConverter<int, TLink> _int32ToAddressConverter =
            ↪  UncheckedConverter<int, TLink>.Default;
21
22          protected readonly TLink Break;
23          protected readonly TLink Continue;
24          protected readonly byte* Links;
25          protected readonly byte* Header;
26
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          protected LinksAvlBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
            ↪  byte* header)
29          {
30              Links = links;
31              Header = header;
32              Break = constants.Break;
33              Continue = constants.Continue;
34          }
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          protected abstract TLink GetTreeRoot();
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          protected abstract TLink GetBasePartValue(TLink link);
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
            ↪  rootSource, TLink rootTarget);
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
            ↪  rootSource, TLink rootTarget);
47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
            ↪  AsRef<LinksHeader<TLink>>(Header);
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
            ↪  AsRef<RawLink<TLink>>(Links + (RawLink<TLink>.SizeInBytes *
            ↪  _addressToInt64Converter.Convert(link)));
53
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
56          {
57              ref var link = ref GetLinkReference(linkIndex);
58              return new Link<TLink>(linkIndex, link.Source, link.Target);
59          }
60
61          [MethodImpl(MethodImplOptions.AggressiveInlining)]
62          protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
63          {
```

```csharp
                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
```

```csharp
137                {
138                    var value = _addressToInt32Converter.Convert(Bit<TLink>.PartialRead(storedValue,
                        ↪  0, 3));
139                    value |= 0xF8 * ((value & 4) >> 2); // if negative, then continue ones to the
                        ↪  end of sbyte
140                    return (sbyte)value;
141                }
142            }
143
144            [MethodImpl(MethodImplOptions.AggressiveInlining)]
145            protected virtual void SetBalanceValue(ref TLink storedValue, sbyte value)
146            {
147                unchecked
148                {
149                    var packagedValue = _int32ToAddressConverter.Convert((byte)value >> 5 & 4 |
                        ↪  value & 3);
150                    var modified = Bit<TLink>.PartialWrite(storedValue, packagedValue, 0, 3);
151                    storedValue = modified;
152                }
153            }
154
155            public TLink this[TLink index]
156            {
157                [MethodImpl(MethodImplOptions.AggressiveInlining)]
158                get
159                {
160                    var root = GetTreeRoot();
161                    if (GreaterOrEqualThan(index, GetSize(root)))
162                    {
163                        return Zero;
164                    }
165                    while (!EqualToZero(root))
166                    {
167                        var left = GetLeftOrDefault(root);
168                        var leftSize = GetSizeOrZero(left);
169                        if (LessThan(index, leftSize))
170                        {
171                            root = left;
172                            continue;
173                        }
174                        if (AreEqual(index, leftSize))
175                        {
176                            return root;
177                        }
178                        root = GetRightOrDefault(root);
179                        index = Subtract(index, Increment(leftSize));
180                    }
181                    return Zero; // TODO: Impossible situation exception (only if tree structure
                        ↪  broken)
182                }
183            }
184
185            /// <summary>
186            /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
                ↪  (концом).
187            /// </summary>
188            /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
189            /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
190            /// <returns>Индекс искомой связи.</returns>
191            [MethodImpl(MethodImplOptions.AggressiveInlining)]
192            public TLink Search(TLink source, TLink target)
193            {
194                var root = GetTreeRoot();
195                while (!EqualToZero(root))
196                {
197                    ref var rootLink = ref GetLinkReference(root);
198                    var rootSource = rootLink.Source;
199                    var rootTarget = rootLink.Target;
200                    if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                        ↪  node.Key < root.Key
201                    {
202                        root = GetLeftOrDefault(root);
203                    }
204                    else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
                        ↪  node.Key > root.Key
205                    {
206                        root = GetRightOrDefault(root);
207                    }
```

```csharp
                    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))
                    {
```

```
287                     return Break;
288                 }
289                 current = GetNext(current);
290                 if (EqualToZero(current) || !AreEqual(GetBasePartValue(current), link))
291                 {
292                     break;
293                 }
294             }
295         }
296         return Continue;
297     }
298
299     [MethodImpl(MethodImplOptions.AggressiveInlining)]
300     protected override void PrintNodeValue(TLink node, StringBuilder sb)
301     {
302         ref var link = ref GetLinkReference(node);
303         sb.Append(' ');
304         sb.Append(link.Source);
305         sb.Append('-');
306         sb.Append('>');
307         sb.Append(link.Target);
308     }
309   }
310 }
```

## 1.80 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksRecursionlessSizeBalancedTreeMethodsBase

```
1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Converters;
7  using static System.Runtime.CompilerServices.Unsafe;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets.Memory.United.Generic
12 {
13     public unsafe abstract class LinksRecursionlessSizeBalancedTreeMethodsBase<TLink> :
         ↪ RecursionlessSizeBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
14     {
15         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
             ↪ UncheckedConverter<TLink, long>.Default;
16
17         protected readonly TLink Break;
18         protected readonly TLink Continue;
19         protected readonly byte* Links;
20         protected readonly byte* Header;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected LinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants,
             ↪ byte* links, byte* header)
24         {
25             Links = links;
26             Header = header;
27             Break = constants.Break;
28             Continue = constants.Continue;
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected abstract TLink GetTreeRoot();
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected abstract TLink GetBasePartValue(TLink link);
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
             ↪ rootSource, TLink rootTarget);
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
             ↪ rootSource, TLink rootTarget);
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
             ↪ AsRef<LinksHeader<TLink>>(Header);
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
47          protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
      ↪     AsRef<RawLink<TLink>>(Links + (RawLink<TLink>.SizeInBytes *
      ↪     _addressToInt64Converter.Convert(link)));
48
49          [MethodImpl(MethodImplOptions.AggressiveInlining)]
50          protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
51          {
52              ref var link = ref GetLinkReference(linkIndex);
53              return new Link<TLink>(linkIndex, link.Source, link.Target);
54          }
55
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
58          {
59              ref var firstLink = ref GetLinkReference(first);
60              ref var secondLink = ref GetLinkReference(second);
61              return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
      ↪     secondLink.Source, secondLink.Target);
62          }
63
64          [MethodImpl(MethodImplOptions.AggressiveInlining)]
65          protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
66          {
67              ref var firstLink = ref GetLinkReference(first);
68              ref var secondLink = ref GetLinkReference(second);
69              return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
      ↪     secondLink.Source, secondLink.Target);
70          }
71
72          public TLink this[TLink index]
73          {
74              [MethodImpl(MethodImplOptions.AggressiveInlining)]
75              get
76              {
77                  var root = GetTreeRoot();
78                  if (GreaterOrEqualThan(index, GetSize(root)))
79                  {
80                      return Zero;
81                  }
82                  while (!EqualToZero(root))
83                  {
84                      var left = GetLeftOrDefault(root);
85                      var leftSize = GetSizeOrZero(left);
86                      if (LessThan(index, leftSize))
87                      {
88                          root = left;
89                          continue;
90                      }
91                      if (AreEqual(index, leftSize))
92                      {
93                          return root;
94                      }
95                      root = GetRightOrDefault(root);
96                      index = Subtract(index, Increment(leftSize));
97                  }
98                  return Zero; // TODO: Impossible situation exception (only if tree structure
      ↪     broken)
99              }
100         }
101
102         /// <summary>
103         /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
      ↪     (концом).
104         /// </summary>
105         /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
106         /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
107         /// <returns>Индекс искомой связи.</returns>
108         [MethodImpl(MethodImplOptions.AggressiveInlining)]
109         public TLink Search(TLink source, TLink target)
110         {
111             var root = GetTreeRoot();
112             while (!EqualToZero(root))
113             {
114                 ref var rootLink = ref GetLinkReference(root);
115                 var rootSource = rootLink.Source;
116                 var rootTarget = rootLink.Target;
117                 if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
      ↪     node.Key < root.Key
```

```csharp
118                    {
119                        root = GetLeftOrDefault(root);
120                    }
121                    else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
                        ↪  node.Key > root.Key
122                    {
123                        root = GetRightOrDefault(root);
124                    }
125                    else // node.Key == root.Key
126                    {
127                        return root;
128                    }
129                }
130                return Zero;
131            }
132
133            // TODO: Return indices range instead of references count
134            [MethodImpl(MethodImplOptions.AggressiveInlining)]
135            public TLink CountUsages(TLink link)
136            {
137                var root = GetTreeRoot();
138                var total = GetSize(root);
139                var totalRightIgnore = Zero;
140                while (!EqualToZero(root))
141                {
142                    var @base = GetBasePartValue(root);
143                    if (LessOrEqualThan(@base, link))
144                    {
145                        root = GetRightOrDefault(root);
146                    }
147                    else
148                    {
149                        totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
150                        root = GetLeftOrDefault(root);
151                    }
152                }
153                root = GetTreeRoot();
154                var totalLeftIgnore = Zero;
155                while (!EqualToZero(root))
156                {
157                    var @base = GetBasePartValue(root);
158                    if (GreaterOrEqualThan(@base, link))
159                    {
160                        root = GetLeftOrDefault(root);
161                    }
162                    else
163                    {
164                        totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
165                        root = GetRightOrDefault(root);
166                    }
167                }
168                return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
169            }
170
171            [MethodImpl(MethodImplOptions.AggressiveInlining)]
172            public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
                ↪  EachUsageCore(@base, GetTreeRoot(), handler);
173
174            // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
                ↪  low-level MSIL stack.
175            [MethodImpl(MethodImplOptions.AggressiveInlining)]
176            private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
177            {
178                var @continue = Continue;
179                if (EqualToZero(link))
180                {
181                    return @continue;
182                }
183                var linkBasePart = GetBasePartValue(link);
184                var @break = Break;
185                if (GreaterThan(linkBasePart, @base))
186                {
187                    if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
188                    {
189                        return @break;
190                    }
191                }
192                else if (LessThan(linkBasePart, @base))
193                {
```

```
194            if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
195            {
196                return @break;
197            }
198        }
199        else //if (linkBasePart == @base)
200        {
201            if (AreEqual(handler(GetLinkValues(link)), @break))
202            {
203                return @break;
204            }
205            if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
206            {
207                return @break;
208            }
209            if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
210            {
211                return @break;
212            }
213        }
214        return @continue;
215    }
216
217    [MethodImpl(MethodImplOptions.AggressiveInlining)]
218    protected override void PrintNodeValue(TLink node, StringBuilder sb)
219    {
220        ref var link = ref GetLinkReference(node);
221        sb.Append(' ');
222        sb.Append(link.Source);
223        sb.Append('-');
224        sb.Append('>');
225        sb.Append(link.Target);
226    }
227    }
228 }
```

## 1.81  ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSizeBalancedTreeMethodsBase.cs

```
1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Converters;
7  using static System.Runtime.CompilerServices.Unsafe;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets.Memory.United.Generic
12 {
13    public unsafe abstract class LinksSizeBalancedTreeMethodsBase<TLink> :
    ↪  SizeBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
14    {
15        private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
        ↪  UncheckedConverter<TLink, long>.Default;
16
17        protected readonly TLink Break;
18        protected readonly TLink Continue;
19        protected readonly byte* Links;
20        protected readonly byte* Header;
21
22        [MethodImpl(MethodImplOptions.AggressiveInlining)]
23        protected LinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
        ↪  byte* header)
24        {
25            Links = links;
26            Header = header;
27            Break = constants.Break;
28            Continue = constants.Continue;
29        }
30
31        [MethodImpl(MethodImplOptions.AggressiveInlining)]
32        protected abstract TLink GetTreeRoot();
33
34        [MethodImpl(MethodImplOptions.AggressiveInlining)]
35        protected abstract TLink GetBasePartValue(TLink link);
36
37        [MethodImpl(MethodImplOptions.AggressiveInlining)]
38        protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
        ↪  rootSource, TLink rootTarget);
39
```

```csharp
        [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)
```

```
110              {
111                  var root = GetTreeRoot();
112                  while (!EqualToZero(root))
113                  {
114                      ref var rootLink = ref GetLinkReference(root);
115                      var rootSource = rootLink.Source;
116                      var rootTarget = rootLink.Target;
117                      if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                           ↪ node.Key < root.Key
118                      {
119                          root = GetLeftOrDefault(root);
120                      }
121                      else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
                           ↪ node.Key > root.Key
122                      {
123                          root = GetRightOrDefault(root);
124                      }
125                      else // node.Key == root.Key
126                      {
127                          return root;
128                      }
129                  }
130                  return Zero;
131              }
132
133              // TODO: Return indices range instead of references count
134              [MethodImpl(MethodImplOptions.AggressiveInlining)]
135              public TLink CountUsages(TLink link)
136              {
137                  var root = GetTreeRoot();
138                  var total = GetSize(root);
139                  var totalRightIgnore = Zero;
140                  while (!EqualToZero(root))
141                  {
142                      var @base = GetBasePartValue(root);
143                      if (LessOrEqualThan(@base, link))
144                      {
145                          root = GetRightOrDefault(root);
146                      }
147                      else
148                      {
149                          totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
150                          root = GetLeftOrDefault(root);
151                      }
152                  }
153                  root = GetTreeRoot();
154                  var totalLeftIgnore = Zero;
155                  while (!EqualToZero(root))
156                  {
157                      var @base = GetBasePartValue(root);
158                      if (GreaterOrEqualThan(@base, link))
159                      {
160                          root = GetLeftOrDefault(root);
161                      }
162                      else
163                      {
164                          totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
165                          root = GetRightOrDefault(root);
166                      }
167                  }
168                  return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
169              }
170
171              [MethodImpl(MethodImplOptions.AggressiveInlining)]
172              public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
                 ↪ EachUsageCore(@base, GetTreeRoot(), handler);
173
174              // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
                 ↪ low-level MSIL stack.
175              [MethodImpl(MethodImplOptions.AggressiveInlining)]
176              private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
177              {
178                  var @continue = Continue;
179                  if (EqualToZero(link))
180                  {
181                      return @continue;
182                  }
183                  var linkBasePart = GetBasePartValue(link);
184                  var @break = Break;
```

```csharp
                    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);
        }
    }
}
```

## 1.82 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesAvlBalancedTreeMethods.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 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)]
```

```csharp
            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);

            [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;
            }
        }
    }
```

## 1.83 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesRecursionlessSizeBalancedTreeMeth...

```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 LinksSourcesRecursionlessSizeBalancedTreeMethods<TLink> :
    ↪ LinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksSourcesRecursionlessSizeBalancedTreeMethods(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;
```

```
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          protected override ref TLink GetRightReference(TLink node) => ref
     ↪   GetLinkReference(node).RightAsSource;

17
18          [MethodImpl(MethodImplOptions.AggressiveInlining)]
19          protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsSource;
20
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsSource;
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          protected override void SetLeft(TLink node, TLink left) =>
     ↪   GetLinkReference(node).LeftAsSource = left;

26
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          protected override void SetRight(TLink node, TLink right) =>
     ↪   GetLinkReference(node).RightAsSource = right;

29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsSource;
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          protected override void SetSize(TLink node, TLink size) =>
     ↪   GetLinkReference(node).SizeAsSource = size;

35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          protected override TLink GetTreeRoot() => GetHeaderReference().RootAsSource;
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
     ↪   TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
     ↪   (AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget));

44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
     ↪   TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
     ↪   (AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));

47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          protected override void ClearNode(TLink node)
50          {
51              ref var link = ref GetLinkReference(node);
52              link.LeftAsSource = Zero;
53              link.RightAsSource = Zero;
54              link.SizeAsSource = Zero;
55          }
56      }
57  }
```

## 1.84 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesSizeBalancedTreeMethods.cs

```
1   using System.Runtime.CompilerServices;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Data.Doublets.Memory.United.Generic
6   {
7       public unsafe class LinksSourcesSizeBalancedTreeMethods<TLink> :
     ↪   LinksSizeBalancedTreeMethodsBase<TLink>
8       {
9           [MethodImpl(MethodImplOptions.AggressiveInlining)]
10          public LinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
     ↪   byte* header) : base(constants, links, header) { }

11
12          [MethodImpl(MethodImplOptions.AggressiveInlining)]
13          protected override ref TLink GetLeftReference(TLink node) => ref
     ↪   GetLinkReference(node).LeftAsSource;

14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          protected override ref TLink GetRightReference(TLink node) => ref
     ↪   GetLinkReference(node).RightAsSource;

17
18          [MethodImpl(MethodImplOptions.AggressiveInlining)]
19          protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsSource;
20
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
        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;
        }
    }
}
```

## 1.85   ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsAvlBalancedTreeMethods.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 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;
```

```csharp
        [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;
        }
    }
}
```

## 1.86 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsRecursionlessSizeBalancedTreeMetho

```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 LinksTargetsRecursionlessSizeBalancedTreeMethods<TLink> :
    ↪  LinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksTargetsRecursionlessSizeBalancedTreeMethods(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;
```

```
15        [MethodImpl(MethodImplOptions.AggressiveInlining)]
16        protected override ref TLink GetRightReference(TLink node) => ref
   ↪    GetLinkReference(node).RightAsTarget;
17
18        [MethodImpl(MethodImplOptions.AggressiveInlining)]
19        protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsTarget;
20
21        [MethodImpl(MethodImplOptions.AggressiveInlining)]
22        protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsTarget;
23
24        [MethodImpl(MethodImplOptions.AggressiveInlining)]
25        protected override void SetLeft(TLink node, TLink left) =>
   ↪    GetLinkReference(node).LeftAsTarget = left;
26
27        [MethodImpl(MethodImplOptions.AggressiveInlining)]
28        protected override void SetRight(TLink node, TLink right) =>
   ↪    GetLinkReference(node).RightAsTarget = right;
29
30        [MethodImpl(MethodImplOptions.AggressiveInlining)]
31        protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsTarget;
32
33        [MethodImpl(MethodImplOptions.AggressiveInlining)]
34        protected override void SetSize(TLink node, TLink size) =>
   ↪    GetLinkReference(node).SizeAsTarget = size;
35
36        [MethodImpl(MethodImplOptions.AggressiveInlining)]
37        protected override TLink GetTreeRoot() => GetHeaderReference().RootAsTarget;
38
39        [MethodImpl(MethodImplOptions.AggressiveInlining)]
40        protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
41
42        [MethodImpl(MethodImplOptions.AggressiveInlining)]
43        protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
   ↪    TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
   ↪    (AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource));
44
45        [MethodImpl(MethodImplOptions.AggressiveInlining)]
46        protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
   ↪    TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
   ↪    (AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource));
47
48        [MethodImpl(MethodImplOptions.AggressiveInlining)]
49        protected override void ClearNode(TLink node)
50        {
51            ref var link = ref GetLinkReference(node);
52            link.LeftAsTarget = Zero;
53            link.RightAsTarget = Zero;
54            link.SizeAsTarget = Zero;
55        }
56    }
57 }
```

## 1.87 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsSizeBalancedTreeMethods.cs

```
1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Memory.United.Generic
6  {
7      public unsafe class LinksTargetsSizeBalancedTreeMethods<TLink> :
   ↪    LinksSizeBalancedTreeMethodsBase<TLink>
8      {
9          [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         public LinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
   ↪    byte* header) : base(constants, links, header) { }
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         protected override ref TLink GetLeftReference(TLink node) => ref
   ↪    GetLinkReference(node).LeftAsTarget;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         protected override ref TLink GetRightReference(TLink node) => ref
   ↪    GetLinkReference(node).RightAsTarget;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsTarget;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsTarget;
```

```csharp
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          protected override void SetLeft(TLink node, TLink left) =>
    ↪   GetLinkReference(node).LeftAsTarget = left;
26
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          protected override void SetRight(TLink node, TLink right) =>
    ↪   GetLinkReference(node).RightAsTarget = right;
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsTarget;
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          protected override void SetSize(TLink node, TLink size) =>
    ↪   GetLinkReference(node).SizeAsTarget = size;
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          protected override TLink GetTreeRoot() => GetHeaderReference().RootAsTarget;
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
    ↪   TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
    ↪   (AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource));
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
    ↪   TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
    ↪   (AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource));
47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          protected override void ClearNode(TLink node)
50          {
51              ref var link = ref GetLinkReference(node);
52              link.LeftAsTarget = Zero;
53              link.RightAsTarget = Zero;
54              link.SizeAsTarget = Zero;
55          }
56      }
57  }
```

## 1.88  ./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinks.cs

```csharp
1   using System;
2   using System.Runtime.CompilerServices;
3   using Platform.Singletons;
4   using Platform.Memory;
5   using static System.Runtime.CompilerServices.Unsafe;
6
7   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9   namespace Platform.Data.Doublets.Memory.United.Generic
10  {
11      public unsafe class UnitedMemoryLinks<TLink> : UnitedMemoryLinksBase<TLink>
12      {
13          private readonly Func<ILinksTreeMethods<TLink>> _createSourceTreeMethods;
14          private readonly Func<ILinksTreeMethods<TLink>> _createTargetTreeMethods;
15          private byte* _header;
16          private byte* _links;
17
18          [MethodImpl(MethodImplOptions.AggressiveInlining)]
19          public UnitedMemoryLinks(string address) : this(address, DefaultLinksSizeStep) { }
20
21          /// <summary>
22          /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
    ↪   минимальным шагом расширения базы данных.
23          /// </summary>
24          /// <param name="address">Полный пусть к файлу базы данных.</param>
25          /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
    ↪   байтах.</param>
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          public UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
    ↪   FileMappedResizableDirectMemory(address, memoryReservationStep),
    ↪   memoryReservationStep) { }
28
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          public UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
    ↪   DefaultLinksSizeStep) { }
```

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnitedMemoryLinks(IResizableDirectMemory memory, long memoryReservationStep,
        ↪  LinksConstants<TLink> constants, IndexTreeType indexTreeType) : base(memory,
        ↪  memoryReservationStep, constants)
        {
            if (indexTreeType == IndexTreeType.SizedAndThreadedAVLBalancedTree)
            {
                _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)));
    }
}
```

## 1.89  ./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinksBase.cs

```csharp
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;
```

```csharp
        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)
        {
```

```csharp
            // 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();
                    }
```

```csharp
                        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))
                        {
```

```csharp
                        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) ||
```

```csharp
                    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);
            }
            freeLink = header.AllocatedLinks = Increment(header.AllocatedLinks);
            _memory.UsedCapacity += LinkSizeInBytes;
        }
```

```csharp
393             return freeLink;
394         }
395
396         [MethodImpl(MethodImplOptions.AggressiveInlining)]
397         public virtual void Delete(IList<TLink> restrictions)
398         {
399             ref var header = ref GetHeaderReference();
400             var link = restrictions[Constants.IndexPart];
401             if (LessThan(link, header.AllocatedLinks))
402             {
403                 UnusedLinksListMethods.AttachAsFirst(link);
404             }
405             else if (AreEqual(link, header.AllocatedLinks))
406             {
407                 header.AllocatedLinks = Decrement(header.AllocatedLinks);
408                 _memory.UsedCapacity -= LinkSizeInBytes;
409                 // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
                 //   пока не дойдём до первой существующей связи
410                 // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
411                 while (GreaterThan(header.AllocatedLinks, GetZero()) &&
                 //   IsUnusedLink(header.AllocatedLinks))
412                 {
413                     UnusedLinksListMethods.Detach(header.AllocatedLinks);
414                     header.AllocatedLinks = Decrement(header.AllocatedLinks);
415                     _memory.UsedCapacity -= LinkSizeInBytes;
416                 }
417             }
418         }
419
420         [MethodImpl(MethodImplOptions.AggressiveInlining)]
421         public IList<TLink> GetLinkStruct(TLink linkIndex)
422         {
423             ref var link = ref GetLinkReference(linkIndex);
424             return new Link<TLink>(linkIndex, link.Source, link.Target);
425         }
426
427         /// <remarks>
428         /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
             ///   адрес реально поменялся
429         ///
430         /// Указатель this.links может быть в том же месте,
431         /// так как 0-я связь не используется и имеет такой же размер как Header,
432         /// поэтому header размещается в том же месте, что и 0-я связь
433         /// </remarks>
434         [MethodImpl(MethodImplOptions.AggressiveInlining)]
435         protected abstract void SetPointers(IResizableDirectMemory memory);
436
437         [MethodImpl(MethodImplOptions.AggressiveInlining)]
438         protected virtual void ResetPointers()
439         {
440             SourcesTreeMethods = null;
441             TargetsTreeMethods = null;
442             UnusedLinksListMethods = null;
443         }
444
445         [MethodImpl(MethodImplOptions.AggressiveInlining)]
446         protected abstract ref LinksHeader<TLink> GetHeaderReference();
447
448         [MethodImpl(MethodImplOptions.AggressiveInlining)]
449         protected abstract ref RawLink<TLink> GetLinkReference(TLink linkIndex);
450
451         [MethodImpl(MethodImplOptions.AggressiveInlining)]
452         protected virtual bool Exists(TLink link)
453             => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
454             && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
455             && !IsUnusedLink(link);
456
457         [MethodImpl(MethodImplOptions.AggressiveInlining)]
458         protected virtual bool IsUnusedLink(TLink linkIndex)
459         {
460             if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
                 //   is not needed
461             {
462                 ref var link = ref GetLinkReference(linkIndex);
463                 return AreEqual(link.SizeAsSource, default) && !AreEqual(link.Source, default);
464             }
465             else
466             {
467                 return true;
```

```csharp
            }
        }

        [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
    }
}
```

## 1.90   ./csharp/Platform.Data.Doublets/Memory/United/Generic/UnusedLinksListMethods.cs

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

```

```csharp
#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> :
        AbsoluteCircularDoublyLinkedListMethods<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;
    }
}
```

## 1.91   ./csharp/Platform.Data.Doublets/Memory/United/RawLink.cs

```csharp
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;
```

```csharp
        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);
    }
}
```

## 1.92 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksRecursionlessSizeBalancedTreeMetho

```csharp
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 UInt32LinksRecursionlessSizeBalancedTreeMethodsBase :
    ↪  LinksRecursionlessSizeBalancedTreeMethodsBase<uint>
    {
        protected new readonly RawLink<uint>* Links;
        protected new readonly LinksHeader<uint>* Header;

        protected UInt32LinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<uint>
        ↪  constants, RawLink<uint>* links, LinksHeader<uint>* header)
            : base(constants, (byte*)links, (byte*)header)
        {
            Links = links;
            Header = header;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetZero() => 0U;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(uint first, uint 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(uint first, uint 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<uint> GetHeaderReference() => ref *Header;

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

## 1.93 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksSizeBalancedTreeMethodsBase.cs

```csharp
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 UInt32LinksSizeBalancedTreeMethodsBase :
    ↪  LinksSizeBalancedTreeMethodsBase<uint>
    {
        protected new readonly RawLink<uint>* Links;
        protected new readonly LinksHeader<uint>* Header;

        protected UInt32LinksSizeBalancedTreeMethodsBase(LinksConstants<uint> constants,
        ↪  RawLink<uint>* links, LinksHeader<uint>* header)
            : base(constants, (byte*)links, (byte*)header)
        {
            Links = links;
            Header = header;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
        protected override uint GetZero() => 0U;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(uint first, uint 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(uint first, uint 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<uint> GetHeaderReference() => ref *Header;

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

## 1.94 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksSourcesRecursionlessSizeBalancedTr

```csharp
using System.Runtime.CompilerServices;

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

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

```csharp
{
    public unsafe class UInt32LinksSourcesRecursionlessSizeBalancedTreeMethods :
        UInt32LinksRecursionlessSizeBalancedTreeMethodsBase
    {
        public UInt32LinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<uint>
            constants, RawLink<uint>* links, LinksHeader<uint>* header) : base(constants, links,
            header) { }

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

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

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

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

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

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

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

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

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

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

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

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

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

## 1.95   ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksSourcesSizeBalancedTreeMethods.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.Specific
{
    public unsafe class UInt32LinksSourcesSizeBalancedTreeMethods :
        UInt32LinksSizeBalancedTreeMethodsBase
    {
        public UInt32LinksSourcesSizeBalancedTreeMethods(LinksConstants<uint> constants,
            RawLink<uint>* links, LinksHeader<uint>* header) : base(constants, links, header) { }

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

```
14      [MethodImpl(MethodImplOptions.AggressiveInlining)]
15      protected override ref uint GetRightReference(uint node) => ref
    ↪  Links[node].RightAsSource;

16
17      [MethodImpl(MethodImplOptions.AggressiveInlining)]
18      protected override uint GetLeft(uint node) => Links[node].LeftAsSource;

19
20      [MethodImpl(MethodImplOptions.AggressiveInlining)]
21      protected override uint GetRight(uint node) => Links[node].RightAsSource;

22
23      [MethodImpl(MethodImplOptions.AggressiveInlining)]
24      protected override void SetLeft(uint node, uint left) => Links[node].LeftAsSource = left;

25
26      [MethodImpl(MethodImplOptions.AggressiveInlining)]
27      protected override void SetRight(uint node, uint right) => Links[node].RightAsSource =
    ↪  right;

28
29      [MethodImpl(MethodImplOptions.AggressiveInlining)]
30      protected override uint GetSize(uint node) => Links[node].SizeAsSource;

31
32      [MethodImpl(MethodImplOptions.AggressiveInlining)]
33      protected override void SetSize(uint node, uint size) => Links[node].SizeAsSource = size;

34
35      [MethodImpl(MethodImplOptions.AggressiveInlining)]
36      protected override uint GetTreeRoot() => Header->RootAsSource;

37
38      [MethodImpl(MethodImplOptions.AggressiveInlining)]
39      protected override uint GetBasePartValue(uint link) => Links[link].Source;

40
41      [MethodImpl(MethodImplOptions.AggressiveInlining)]
42      protected override bool FirstIsToTheLeftOfSecond(uint firstSource, uint firstTarget,
    ↪  uint secondSource, uint secondTarget)
43          => firstSource < secondSource || (firstSource == secondSource && firstTarget <
        ↪  secondTarget);

44
45      [MethodImpl(MethodImplOptions.AggressiveInlining)]
46      protected override bool FirstIsToTheRightOfSecond(uint firstSource, uint firstTarget,
    ↪  uint secondSource, uint secondTarget)
47          => firstSource > secondSource || (firstSource == secondSource && firstTarget >
        ↪  secondTarget);

48
49      [MethodImpl(MethodImplOptions.AggressiveInlining)]
50      protected override void ClearNode(uint node)
51      {
52          ref var link = ref Links[node];
53          link.LeftAsSource = 0U;
54          link.RightAsSource = 0U;
55          link.SizeAsSource = 0U;
56      }
57      }
58  }
```

## 1.96 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksTargetsRecursionlessSizeBalancedTre

```
1   using System.Runtime.CompilerServices;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Data.Doublets.Memory.United.Specific
6   {
7       public unsafe class UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods :
    ↪  UInt32LinksRecursionlessSizeBalancedTreeMethodsBase
8       {
9           public UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<uint>
        ↪  constants, RawLink<uint>* links, LinksHeader<uint>* header) : base(constants, links,
        ↪  header) { }
10
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          protected override ref uint GetLeftReference(uint node) => ref Links[node].LeftAsTarget;
13
14          [MethodImpl(MethodImplOptions.AggressiveInlining)]
15          protected override ref uint GetRightReference(uint node) => ref
        ↪  Links[node].RightAsTarget;

16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          protected override uint GetLeft(uint node) => Links[node].LeftAsTarget;

19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          protected override uint GetRight(uint node) => Links[node].RightAsTarget;

22
```

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

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

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

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

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

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

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

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

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

## 1.97 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksTargetsSizeBalancedTreeMethods.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.Specific
{
    public unsafe class UInt32LinksTargetsSizeBalancedTreeMethods :
    →  UInt32LinksSizeBalancedTreeMethodsBase
    {
        public UInt32LinksTargetsSizeBalancedTreeMethods(LinksConstants<uint> constants,
        →  RawLink<uint>* links, LinksHeader<uint>* header) : base(constants, links, header) { }

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
33    protected override void SetSize(uint node, uint size) => Links[node].SizeAsTarget = size;
34
35    [MethodImpl(MethodImplOptions.AggressiveInlining)]
36    protected override uint GetTreeRoot() => Header->RootAsTarget;
37
38    [MethodImpl(MethodImplOptions.AggressiveInlining)]
39    protected override uint GetBasePartValue(uint link) => Links[link].Target;
40
41    [MethodImpl(MethodImplOptions.AggressiveInlining)]
42    protected override bool FirstIsToTheLeftOfSecond(uint firstSource, uint firstTarget,
    ↪   uint secondSource, uint secondTarget)
43        => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
        ↪   secondSource);
44
45    [MethodImpl(MethodImplOptions.AggressiveInlining)]
46    protected override bool FirstIsToTheRightOfSecond(uint firstSource, uint firstTarget,
    ↪   uint secondSource, uint secondTarget)
47        => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
        ↪   secondSource);
48
49    [MethodImpl(MethodImplOptions.AggressiveInlining)]
50    protected override void ClearNode(uint node)
51    {
52        ref var link = ref Links[node];
53        link.LeftAsTarget = 0U;
54        link.RightAsTarget = 0U;
55        link.SizeAsTarget = 0U;
56    }
57    }
58 }
```

## 1.98 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32UnitedMemoryLinks.cs

```csharp
1  using System;
2  using System.Runtime.CompilerServices;
3  using Platform.Memory;
4  using Platform.Singletons;
5  using Platform.Data.Doublets.Memory.United.Generic;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Memory.United.Specific
10 {
11     /// <summary>
12     /// <para>Represents a low-level implementation of direct access to resizable memory, for
        ↪   organizing the storage of links with addresses represented as <see cref="uint" />.</para>
13     /// <para>Представляет низкоуровневую реализация прямого доступа к памяти с переменным
        ↪   размером, для организации хранения связей с адресами представленными в виде <see
        ↪   cref="uint"/>.</para>
14     /// </summary>
15     public unsafe class UInt32UnitedMemoryLinks : UnitedMemoryLinksBase<uint>
16     {
17         private readonly Func<ILinksTreeMethods<uint>> _createSourceTreeMethods;
18         private readonly Func<ILinksTreeMethods<uint>> _createTargetTreeMethods;
19         private LinksHeader<uint>* _header;
20         private RawLink<uint>* _links;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public UInt32UnitedMemoryLinks(string address) : this(address, DefaultLinksSizeStep) { }
24
25         /// <summary>
26         /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
        ↪   минимальным шагом расширения базы данных.
27         /// </summary>
28         /// <param name="address">Полный пусть к файлу базы данных.</param>
29         /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
        ↪   байтах.</param>
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public UInt32UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
        ↪   FileMappedResizableDirectMemory(address, memoryReservationStep),
        ↪   memoryReservationStep) { }
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public UInt32UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
        ↪   DefaultLinksSizeStep) { }
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public UInt32UnitedMemoryLinks(IResizableDirectMemory memory, long
        ↪   memoryReservationStep) : this(memory, memoryReservationStep,
        ↪   Default<LinksConstants<uint>>.Instance, IndexTreeType.Default) { }
```

```csharp
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32UnitedMemoryLinks(IResizableDirectMemory memory, long
        ↪  memoryReservationStep, LinksConstants<uint> constants, IndexTreeType indexTreeType)
        ↪  : base(memory, memoryReservationStep, constants)
        {
            if (indexTreeType == IndexTreeType.SizeBalancedTree)
            {
                _createSourceTreeMethods = () => new
                    ↪  UInt32LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
                _createTargetTreeMethods = () => new
                    ↪  UInt32LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
            }
            else
            {
                _createSourceTreeMethods = () => new
                    ↪  UInt32LinksSourcesRecursionlessSizeBalancedTreeMethods(Constants, _links,
                    ↪  _header);
                _createTargetTreeMethods = () => new
                    ↪  UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods(Constants, _links,
                    ↪  _header);
            }
            Init(memory, memoryReservationStep);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetPointers(IResizableDirectMemory memory)
        {
            _header = (LinksHeader<uint>*)memory.Pointer;
            _links = (RawLink<uint>*)memory.Pointer;
            SourcesTreeMethods = _createSourceTreeMethods();
            TargetsTreeMethods = _createTargetTreeMethods();
            UnusedLinksListMethods = new UInt32UnusedLinksListMethods(_links, _header);
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLink<uint> GetLinkReference(uint linkIndex) => ref
        ↪  _links[linkIndex];

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetZero() => 0U;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetOne() => 1U;

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
        protected override uint Add(uint first, uint second) => first + second;

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

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

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

## 1.99 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32UnusedLinksListMethods.cs

```csharp
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 UInt32UnusedLinksListMethods : UnusedLinksListMethods<uint>
    {
        private readonly RawLink<uint>* _links;
        private readonly LinksHeader<uint>* _header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32UnusedLinksListMethods(RawLink<uint>* links, LinksHeader<uint>* header)
            : base((byte*)links, (byte*)header)
        {
            _links = links;
            _header = header;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLink<uint> GetLinkReference(uint link) => ref _links[link];

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref LinksHeader<uint> GetHeaderReference() => ref *_header;
    }
}
```

## 1.100 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksAvlBalancedTreeMethodsBase.cs

```csharp
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() => OUL;

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

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

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

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

```csharp
        [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)]
```

```csharp
                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];
        }
    }
```

```csharp
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 UInt64LinksRecursionlessSizeBalancedTreeMethodsBase :
        LinksRecursionlessSizeBalancedTreeMethodsBase<ulong>
    {
        protected new readonly RawLink<ulong>* Links;
        protected new readonly LinksHeader<ulong>* Header;

        protected UInt64LinksRecursionlessSizeBalancedTreeMethodsBase(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)]
```

```csharp
        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];
    }
}
```

## 1.102 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSizeBalancedTreeMethodsBase.cs

```csharp
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)]
```

```csharp
        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];
    }
}
```

## 1.103   ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesAvlBalancedTreeMethods.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.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;
```

```csharp
28
29            [MethodImpl(MethodImplOptions.AggressiveInlining)]
30            protected override ulong GetSize(ulong node) => GetSizeValue(Links[node].SizeAsSource);
31
32            [MethodImpl(MethodImplOptions.AggressiveInlining)]
33            protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
    ↪   Links[node].SizeAsSource, size);
34
35            [MethodImpl(MethodImplOptions.AggressiveInlining)]
36            protected override bool GetLeftIsChild(ulong node) =>
    ↪   GetLeftIsChildValue(Links[node].SizeAsSource);
37
38            //[MethodImpl(MethodImplOptions.AggressiveInlining)]
39            //protected override bool GetLeftIsChild(ulong node) => IsChild(node, GetLeft(node));
40
41            [MethodImpl(MethodImplOptions.AggressiveInlining)]
42            protected override void SetLeftIsChild(ulong node, bool value) =>
    ↪   SetLeftIsChildValue(ref Links[node].SizeAsSource, value);
43
44            [MethodImpl(MethodImplOptions.AggressiveInlining)]
45            protected override bool GetRightIsChild(ulong node) =>
    ↪   GetRightIsChildValue(Links[node].SizeAsSource);
46
47            //[MethodImpl(MethodImplOptions.AggressiveInlining)]
48            //protected override bool GetRightIsChild(ulong node) => IsChild(node, GetRight(node));
49
50            [MethodImpl(MethodImplOptions.AggressiveInlining)]
51            protected override void SetRightIsChild(ulong node, bool value) =>
    ↪   SetRightIsChildValue(ref Links[node].SizeAsSource, value);
52
53            [MethodImpl(MethodImplOptions.AggressiveInlining)]
54            protected override sbyte GetBalance(ulong node) =>
    ↪   GetBalanceValue(Links[node].SizeAsSource);
55
56            [MethodImpl(MethodImplOptions.AggressiveInlining)]
57            protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
    ↪   Links[node].SizeAsSource, value);
58
59            [MethodImpl(MethodImplOptions.AggressiveInlining)]
60            protected override ulong GetTreeRoot() => Header->RootAsSource;
61
62            [MethodImpl(MethodImplOptions.AggressiveInlining)]
63            protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
64
65            [MethodImpl(MethodImplOptions.AggressiveInlining)]
66            protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
    ↪   ulong secondSource, ulong secondTarget)
67                => firstSource < secondSource || (firstSource == secondSource && firstTarget <
    ↪   secondTarget);
68
69            [MethodImpl(MethodImplOptions.AggressiveInlining)]
70            protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
    ↪   ulong secondSource, ulong secondTarget)
71                => firstSource > secondSource || (firstSource == secondSource && firstTarget >
    ↪   secondTarget);
72
73            [MethodImpl(MethodImplOptions.AggressiveInlining)]
74            protected override void ClearNode(ulong node)
75            {
76                ref var link = ref Links[node];
77                link.LeftAsSource = 0UL;
78                link.RightAsSource = 0UL;
79                link.SizeAsSource = 0UL;
80            }
81        }
82    }
```

## 1.104    ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesRecursionlessSizeBalancedTr

```csharp
1    using System.Runtime.CompilerServices;
2
3    #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5    namespace Platform.Data.Doublets.Memory.United.Specific
6    {
7        public unsafe class UInt64LinksSourcesRecursionlessSizeBalancedTreeMethods :
    ↪   UInt64LinksRecursionlessSizeBalancedTreeMethodsBase
8        {
```

```csharp
        public UInt64LinksSourcesRecursionlessSizeBalancedTreeMethods(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;
        }
    }
}
```

## 1.105   ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesSizeBalancedTreeMethods.c

```csharp
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;
```

```csharp
13
14             [MethodImpl(MethodImplOptions.AggressiveInlining)]
15             protected override ref ulong GetRightReference(ulong node) => ref
   ↪   Links[node].RightAsSource;
16
17             [MethodImpl(MethodImplOptions.AggressiveInlining)]
18             protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;
19
20             [MethodImpl(MethodImplOptions.AggressiveInlining)]
21             protected override ulong GetRight(ulong node) => Links[node].RightAsSource;
22
23             [MethodImpl(MethodImplOptions.AggressiveInlining)]
24             protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
   ↪   left;
25
26             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
   ↪   right;
28
29             [MethodImpl(MethodImplOptions.AggressiveInlining)]
30             protected override ulong GetSize(ulong node) => Links[node].SizeAsSource;
31
32             [MethodImpl(MethodImplOptions.AggressiveInlining)]
33             protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsSource =
   ↪   size;
34
35             [MethodImpl(MethodImplOptions.AggressiveInlining)]
36             protected override ulong GetTreeRoot() => Header->RootAsSource;
37
38             [MethodImpl(MethodImplOptions.AggressiveInlining)]
39             protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
40
41             [MethodImpl(MethodImplOptions.AggressiveInlining)]
42             protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
   ↪   ulong secondSource, ulong secondTarget)
43                 => firstSource < secondSource || (firstSource == secondSource && firstTarget <
   ↪   secondTarget);
44
45             [MethodImpl(MethodImplOptions.AggressiveInlining)]
46             protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
   ↪   ulong secondSource, ulong secondTarget)
47                 => firstSource > secondSource || (firstSource == secondSource && firstTarget >
   ↪   secondTarget);
48
49             [MethodImpl(MethodImplOptions.AggressiveInlining)]
50             protected override void ClearNode(ulong node)
51             {
52                 ref var link = ref Links[node];
53                 link.LeftAsSource = 0UL;
54                 link.RightAsSource = 0UL;
55                 link.SizeAsSource = 0UL;
56             }
57         }
58     }
```

## 1.106   ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsAvlBalancedTreeMethods.cs

```csharp
1   using System.Runtime.CompilerServices;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Data.Doublets.Memory.United.Specific
6   {
7       public unsafe class UInt64LinksTargetsAvlBalancedTreeMethods :
   ↪   UInt64LinksAvlBalancedTreeMethodsBase
8       {
9           public UInt64LinksTargetsAvlBalancedTreeMethods(LinksConstants<ulong> constants,
   ↪   RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
   ↪   { }
10
11             [MethodImpl(MethodImplOptions.AggressiveInlining)]
12             protected override ref ulong GetLeftReference(ulong node) => ref
   ↪   Links[node].LeftAsTarget;
13
14             [MethodImpl(MethodImplOptions.AggressiveInlining)]
15             protected override ref ulong GetRightReference(ulong node) => ref
   ↪   Links[node].RightAsTarget;
16
17             [MethodImpl(MethodImplOptions.AggressiveInlining)]
18             protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;
```

```csharp
                [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;
                }
        }
}
```

## 1.107 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsRecursionlessSizeBalancedTr

```csharp
using System.Runtime.CompilerServices;

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

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

```csharp
    public unsafe class UInt64LinksTargetsRecursionlessSizeBalancedTreeMethods :
    ↪  UInt64LinksRecursionlessSizeBalancedTreeMethodsBase
    {
        public UInt64LinksTargetsRecursionlessSizeBalancedTreeMethods(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;
        }
    }
}
```

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

```csharp
        [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;
        }
    }
}
```

## 1.109   ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnitedMemoryLinks.cs

```csharp
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>
    {
```

```csharp
        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, IndexTreeType.Default) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64UnitedMemoryLinks(IResizableDirectMemory memory, long
        ↪  memoryReservationStep, LinksConstants<ulong> constants, IndexTreeType indexTreeType)
        ↪  : base(memory, memoryReservationStep, constants)
        {
            if (indexTreeType == IndexTreeType.SizedAndThreadedAVLBalancedTree)
            {
                _createSourceTreeMethods = () => new
                ↪  UInt64LinksSourcesAvlBalancedTreeMethods(Constants, _links, _header);
                _createTargetTreeMethods = () => new
                ↪  UInt64LinksTargetsAvlBalancedTreeMethods(Constants, _links, _header);
            }
            else if (indexTreeType == IndexTreeType.SizeBalancedTree)
            {
                _createSourceTreeMethods = () => new
                ↪  UInt64LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
                _createTargetTreeMethods = () => new
                ↪  UInt64LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
            }
            else
            {
                _createSourceTreeMethods = () => new
                ↪  UInt64LinksSourcesRecursionlessSizeBalancedTreeMethods(Constants, _links,
                ↪  _header);
                _createTargetTreeMethods = () => new
                ↪  UInt64LinksTargetsRecursionlessSizeBalancedTreeMethods(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;
        }
```

```csharp
        [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;
    }
}
```

## 1.110 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnusedLinksListMethods.cs

```csharp
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;
    }
}
```

## 1.111 ./csharp/Platform.Data.Doublets/Numbers/Raw/LongRawNumberSequenceToNumberConverter.cs

```csharp
using System.Runtime.CompilerServices;
using Platform.Collections.Stacks;
```

```csharp
using Platform.Converters;
using Platform.Numbers;
using Platform.Reflection;
using Platform.Data.Doublets.Decorators;
using Platform.Data.Doublets.Sequences.Walkers;

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

namespace Platform.Data.Doublets.Numbers.Raw
{
    public class LongRawNumberSequenceToNumberConverter<TSource, TTarget> :
    ↪ LinksDecoratorBase<TSource>, IConverter<TSource, TTarget>
    {
        private static readonly int _bitsPerRawNumber = NumericType<TSource>.BitsSize - 1;
        private static readonly UncheckedConverter<TSource, TTarget> _sourceToTargetConverter =
        ↪ UncheckedConverter<TSource, TTarget>.Default;

        private readonly IConverter<TSource> _numberToAddressConverter;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LongRawNumberSequenceToNumberConverter(ILinks<TSource> links, IConverter<TSource>
        ↪ numberToAddressConverter) : base(links) => _numberToAddressConverter =
        ↪ numberToAddressConverter;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TTarget Convert(TSource source)
        {
            var constants = Links.Constants;
            var externalReferencesRange = constants.ExternalReferencesRange;
            if (externalReferencesRange.HasValue &&
            ↪ externalReferencesRange.Value.Contains(source))
            {
                return
                ↪ _sourceToTargetConverter.Convert(_numberToAddressConverter.Convert(source));
            }
            else
            {
                var pair = Links.GetLink(source);
                var walker = new LeftSequenceWalker<TSource>(Links, new DefaultStack<TSource>(),
                ↪ (link) => externalReferencesRange.HasValue &&
                ↪ externalReferencesRange.Value.Contains(link));
                TTarget result = default;
                foreach (var element in walker.Walk(source))
                {
                    result = Bit.Or(Bit.ShiftLeft(result, _bitsPerRawNumber), Convert(element));
                }
                return result;
            }
        }
    }
}
```

## 1.112 ./csharp/Platform.Data.Doublets/Numbers/Raw/NumberToLongRawNumberSequenceConverter.cs

```csharp
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Converters;
using Platform.Numbers;
using Platform.Reflection;
using Platform.Data.Doublets.Decorators;

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

namespace Platform.Data.Doublets.Numbers.Raw
{
    public class NumberToLongRawNumberSequenceConverter<TSource, TTarget> :
    ↪ LinksDecoratorBase<TTarget>, IConverter<TSource, TTarget>
    {
        private static readonly Comparer<TSource> _comparer = Comparer<TSource>.Default;
        private static readonly TSource _maximumValue = NumericType<TSource>.MaxValue;
        private static readonly int _bitsPerRawNumber = NumericType<TTarget>.BitsSize - 1;
        private static readonly TSource _bitMask = Bit.ShiftRight(_maximumValue,
        ↪ NumericType<TTarget>.BitsSize + 1);
        private static readonly TSource _maximumConvertableAddress = CheckedConverter<TTarget,
        ↪ TSource>.Default.Convert(Arithmetic.Decrement(Hybrid<TTarget>.ExternalZero));
        private static readonly UncheckedConverter<TSource, TTarget> _sourceToTargetConverter =
        ↪ UncheckedConverter<TSource, TTarget>.Default;

        private readonly IConverter<TTarget> _addressToNumberConverter;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
24          public NumberToLongRawNumberSequenceConverter(ILinks<TTarget> links, IConverter<TTarget>
       ↪  addressToNumberConverter) : base(links) => _addressToNumberConverter =
       ↪  addressToNumberConverter;
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          public TTarget Convert(TSource source)
28          {
29              if (_comparer.Compare(source, _maximumConvertableAddress) > 0)
30              {
31                  var numberPart = Bit.And(source, _bitMask);
32                  var convertedNumber = _addressToNumberConverter.Convert(_sourceToTargetConverter
       ↪  .Convert(numberPart));
33                  return Links.GetOrCreate(convertedNumber, Convert(Bit.ShiftRight(source,
       ↪  _bitsPerRawNumber)));
34              }
35              else
36              {
37                  return
       ↪  _addressToNumberConverter.Convert(_sourceToTargetConverter.Convert(source));
38              }
39          }
40      }
41  }
```

## 1.113 ./csharp/Platform.Data.Doublets/Numbers/Unary/AddressToUnaryNumberConverter.cs

```
1   using System.Collections.Generic;
2   using Platform.Reflection;
3   using Platform.Converters;
4   using Platform.Numbers;
5   using System.Runtime.CompilerServices;
6
7   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9   namespace Platform.Data.Doublets.Numbers.Unary
10  {
11      public class AddressToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
       ↪  IConverter<TLink>
12      {
13          private static readonly EqualityComparer<TLink> _equalityComparer =
       ↪  EqualityComparer<TLink>.Default;
14          private static readonly TLink _zero = default;
15          private static readonly TLink _one = Arithmetic.Increment(_zero);
16
17          private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
       ↪  powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
       ↪  powerOf2ToUnaryNumberConverter;
21
22          [MethodImpl(MethodImplOptions.AggressiveInlining)]
23          public TLink Convert(TLink number)
24          {
25              var links = _links;
26              var nullConstant = links.Constants.Null;
27              var target = nullConstant;
28              for (var i = 0; !_equalityComparer.Equals(number, _zero) && i <
       ↪  NumericType<TLink>.BitsSize; i++)
29              {
30                  if (_equalityComparer.Equals(Bit.And(number, _one), _one))
31                  {
32                      target = _equalityComparer.Equals(target, nullConstant)
33                          ? _powerOf2ToUnaryNumberConverter.Convert(i)
34                          : links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
35                  }
36                  number = Bit.ShiftRight(number, 1);
37              }
38              return target;
39          }
40      }
41  }
```

## 1.114 ./csharp/Platform.Data.Doublets/Numbers/Unary/LinkToItsFrequencyNumberConveter.cs

```
1   using System;
2   using System.Collections.Generic;
3   using Platform.Interfaces;
4   using Platform.Converters;
5   using System.Runtime.CompilerServices;
6
```

```csharp
#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);
        }
    }
}
```

## 1.115 ./csharp/Platform.Data.Doublets/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs

```csharp
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);
```

```
33            var powerOf2 = _links.GetOrCreate(previousPowerOf2, previousPowerOf2);
34            _unaryNumberPowersOf2[power] = powerOf2;
35            return powerOf2;
36        }
37    }
38 }
```

## 1.116 ./csharp/Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressAddOperationConverter.cs

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Converters;
4  using Platform.Numbers;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Numbers.Unary
9  {
10     public class UnaryNumberToAddressAddOperationConverter<TLink> : LinksOperatorBase<TLink>,
       ↪  IConverter<TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
13         private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
           ↪  UncheckedConverter<TLink, ulong>.Default;
14         private static readonly UncheckedConverter<ulong, TLink> _uInt64ToAddressConverter =
           ↪  UncheckedConverter<ulong, TLink>.Default;
15         private static readonly TLink _zero = default;
16         private static readonly TLink _one = Arithmetic.Increment(_zero);
17
18         private readonly Dictionary<TLink, TLink> _unaryToUInt64;
19         private readonly TLink _unaryOne;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public UnaryNumberToAddressAddOperationConverter(ILinks<TLink> links, TLink unaryOne)
23             : base(links)
24         {
25             _unaryOne = unaryOne;
26             _unaryToUInt64 = CreateUnaryToUInt64Dictionary(links, unaryOne);
27         }
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public TLink Convert(TLink unaryNumber)
31         {
32             if (_equalityComparer.Equals(unaryNumber, default))
33             {
34                 return default;
35             }
36             if (_equalityComparer.Equals(unaryNumber, _unaryOne))
37             {
38                 return _one;
39             }
40             var links = _links;
41             var source = links.GetSource(unaryNumber);
42             var target = links.GetTarget(unaryNumber);
43             if (_equalityComparer.Equals(source, target))
44             {
45                 return _unaryToUInt64[unaryNumber];
46             }
47             else
48             {
49                 var result = _unaryToUInt64[source];
50                 TLink lastValue;
51                 while (!_unaryToUInt64.TryGetValue(target, out lastValue))
52                 {
53                     source = links.GetSource(target);
54                     result = Arithmetic<TLink>.Add(result, _unaryToUInt64[source]);
55                     target = links.GetTarget(target);
56                 }
57                 result = Arithmetic<TLink>.Add(result, lastValue);
58                 return result;
59             }
60         }
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         private static Dictionary<TLink, TLink> CreateUnaryToUInt64Dictionary(ILinks<TLink>
           ↪  links, TLink unaryOne)
64         {
65             var unaryToUInt64 = new Dictionary<TLink, TLink>
66             {
67                 { unaryOne, _one }
```

```
68              };
69              var unary = unaryOne;
70              var number = _one;
71              for (var i = 1; i < 64; i++)
72              {
73                  unary = links.GetOrCreate(unary, unary);
74                  number = Double(number);
75                  unaryToUInt64.Add(unary, number);
76              }
77              return unaryToUInt64;
78          }
79
80          [MethodImpl(MethodImplOptions.AggressiveInlining)]
81          private static TLink Double(TLink number) =>
       ↪    _uInt64ToAddressConverter.Convert(_addressToUInt64Converter.Convert(number) * 2UL);
82      }
83  }
```

## 1.117 ./csharp/Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressOrOperationConverter.cs

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Reflection;
4   using Platform.Converters;
5   using Platform.Numbers;
6
7   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9   namespace Platform.Data.Doublets.Numbers.Unary
10  {
11      public class UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
        ↪   IConverter<TLink>
12      {
13          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪   EqualityComparer<TLink>.Default;
14          private static readonly TLink _zero = default;
15          private static readonly TLink _one = Arithmetic.Increment(_zero);
16
17          private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
            ↪   TLink> powerOf2ToUnaryNumberConverter) : base(links) => _unaryNumberPowerOf2Indicies
            ↪   = CreateUnaryNumberPowerOf2IndiciesDictionary(powerOf2ToUnaryNumberConverter);
21
22          [MethodImpl(MethodImplOptions.AggressiveInlining)]
23          public TLink Convert(TLink sourceNumber)
24          {
25              var links = _links;
26              var nullConstant = links.Constants.Null;
27              var source = sourceNumber;
28              var target = nullConstant;
29              if (!_equalityComparer.Equals(source, nullConstant))
30              {
31                  while (true)
32                  {
33                      if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
34                      {
35                          SetBit(ref target, powerOf2Index);
36                          break;
37                      }
38                      else
39                      {
40                          powerOf2Index = _unaryNumberPowerOf2Indicies[links.GetSource(source)];
41                          SetBit(ref target, powerOf2Index);
42                          source = links.GetTarget(source);
43                      }
44                  }
45              }
46              return target;
47          }
48
49          [MethodImpl(MethodImplOptions.AggressiveInlining)]
50          private static Dictionary<TLink, int>
            ↪   CreateUnaryNumberPowerOf2IndiciesDictionary(IConverter<int, TLink>
            ↪   powerOf2ToUnaryNumberConverter)
51          {
52              var unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
53              for (int i = 0; i < NumericType<TLink>.BitsSize; i++)
54              {
55                  unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
```

```
56              }
57                  return unaryNumberPowerOf2Indicies;
58              }
59
60          [MethodImpl(MethodImplOptions.AggressiveInlining)]
61          private static void SetBit(ref TLink target, int powerOf2Index) => target =
     ↪   Bit.Or(target, Bit.ShiftLeft(_one, powerOf2Index));
62      }
63  }
```

## 1.118  ./csharp/Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Interfaces;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.PropertyOperators
8  {
9      public class PropertiesOperator<TLink> : LinksOperatorBase<TLink>, IProperties<TLink, TLink,
     ↪   TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
     ↪   EqualityComparer<TLink>.Default;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public PropertiesOperator(ILinks<TLink> links) : base(links) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public TLink GetValue(TLink @object, TLink property)
18         {
19             var links = _links;
20             var objectProperty = links.SearchOrDefault(@object, property);
21             if (_equalityComparer.Equals(objectProperty, default))
22             {
23                 return default;
24             }
25             var constants = links.Constants;
26             var any = constants.Any;
27             var query = new Link<TLink>(any, objectProperty, any);
28             var valueLink = links.SingleOrDefault(query);
29             if (valueLink == null)
30             {
31                 return default;
32             }
33             return links.GetTarget(valueLink[constants.IndexPart]);
34         }
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public void SetValue(TLink @object, TLink property, TLink value)
38         {
39             var links = _links;
40             var objectProperty = links.GetOrCreate(@object, property);
41             links.DeleteMany(links.AllIndices(links.Constants.Any, objectProperty));
42             links.GetOrCreate(objectProperty, value);
43         }
44     }
45  }
```

## 1.119  ./csharp/Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Interfaces;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.PropertyOperators
8  {
9      public class PropertyOperator<TLink> : LinksOperatorBase<TLink>, IProperty<TLink, TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
     ↪   EqualityComparer<TLink>.Default;
12
13         private readonly TLink _propertyMarker;
14         private readonly TLink _propertyValueMarker;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public PropertyOperator(ILinks<TLink> links, TLink propertyMarker, TLink
     ↪   propertyValueMarker) : base(links)
18         {
```

```csharp
                    _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);
                }
            }
        }
    }
```

## 1.120  ./csharp/Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.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.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)
            {
```

```csharp
                    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];
                }
            }
        }
    }
```

## 1.121 ./csharp/Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs

```csharp
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;
```

```csharp
        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 = new Doublet<TLink>(sequence[i - 1], sequence[i]);
                LinkFrequency<TLink> data;
                if (_doInitialFrequenciesIncrement)
                {
                    data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
                }
```

```csharp
                    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];
```

```
176                         }
177                     }
178                     if (w < newLength)
179                     {
180                         copy[w] = copy[r];
181                     }
182                     oldLength = newLength;
183                     ResetMaxDoublet();
184                     UpdateMaxDoublet(copy, newLength);
185                 }
186                 return newLength;
187             }
188
189             [MethodImpl(MethodImplOptions.AggressiveInlining)]
190             private void ResetMaxDoublet()
191             {
192                 _maxDoublet = new Doublet<TLink>();
193                 _maxDoubletData = new LinkFrequency<TLink>();
194             }
195
196             [MethodImpl(MethodImplOptions.AggressiveInlining)]
197             private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
198             {
199                 Doublet<TLink> doublet = default;
200                 for (var i = 1; i < length; i++)
201                 {
202                     doublet = new Doublet<TLink>(copy[i - 1].Element, copy[i].Element);
203                     UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
204                 }
205             }
206
207             [MethodImpl(MethodImplOptions.AggressiveInlining)]
208             private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
209             {
210                 var frequency = data.Frequency;
211                 var maxFrequency = _maxDoubletData.Frequency;
212                 //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
213                 //    (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
214                 //    compression string data (and gives collisions quickly) */ _maxDoublet.Source +
215                 //    _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) */
                {
216                     _maxDoublet = doublet;
217                     _maxDoubletData = data;
218                 }
219             }
220         }
221     }
```

## 1.122  ./csharp/Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Converters;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.Sequences.Converters
8   {
9       public abstract class LinksListToSequenceConverterBase<TLink> : LinksOperatorBase<TLink>,
            IConverter<IList<TLink>, TLink>
10      {
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          protected LinksListToSequenceConverterBase(ILinks<TLink> links) : base(links) { }
13
14          [MethodImpl(MethodImplOptions.AggressiveInlining)]
15          public abstract TLink Convert(IList<TLink> source);
16      }
17  }
```

## 1.123  ./csharp/Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Collections.Lists;
```

```csharp
using Platform.Converters;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;

#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 OptimalVariantConverter(ILinks<TLink> links, LinkFrequenciesCache<TLink>
            linkFrequenciesCache)
            : this(links, new SequenceToItsLocalElementLevelsConverter<TLink>(links, new Frequen
                ciesCacheBasedLinkToItsFrequencyNumberConverter<TLink>(linkFrequenciesCache))) {
                }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public OptimalVariantConverter(ILinks<TLink> links)
            : this(links, new LinkFrequenciesCache<TLink>(links, new
                TotalSequenceSymbolFrequencyCounter<TLink>(links))) { }

        [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)
                        {
```

```
74              sequence[w] = sequence[i];
75              levels[w] = levels[i];
76              w++;
77          }
78      }
79      else
80      {
81          currentLevel = levels[i];
82          levelRepeat = 1;
83          if (skipOnce)
84          {
85              skipOnce = false;
86          }
87          else
88          {
89              sequence[w] = sequence[i - 1];
90              levels[w] = levels[i - 1];
91              previousLevel = levels[w];
92              w++;
93          }
94          if (i == length - 1)
95          {
96              sequence[w] = sequence[i];
97              levels[w] = levels[i];
98              w++;
99          }
100     }
101 }
102 length = w;
103     }
104     return _links.GetOrCreate(sequence[0], sequence[1]);
105 }
106
107 [MethodImpl(MethodImplOptions.AggressiveInlining)]
108 private static TLink GetGreatestNeigbourLowerThanCurrentOrCurrent(TLink previous, TLink
    ↪  current, TLink next)
109 {
110     return _comparer.Compare(previous, next) > 0
111         ? _comparer.Compare(previous, current) < 0 ? previous : current
112         : _comparer.Compare(next, current) < 0 ? next : current;
113 }
114
115 [MethodImpl(MethodImplOptions.AggressiveInlining)]
116 private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
    ↪  _comparer.Compare(next, current) < 0 ? next : current;
117
118 [MethodImpl(MethodImplOptions.AggressiveInlining)]
119 private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
    ↪  => _comparer.Compare(previous, current) < 0 ? previous : current;
120     }
121 }
```

## 1.124 ./csharp/Platform.Data.Doublets/Sequences/Converters/SequenceToItsLocalElementLevelsConverter.cs

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Converters;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Sequences.Converters
8  {
9      public class SequenceToItsLocalElementLevelsConverter<TLink> : LinksOperatorBase<TLink>,
        ↪  IConverter<IList<TLink>>
10     {
11         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
12
13         private readonly IConverter<Doublet<TLink>, TLink> _linkToItsFrequencyToNumberConveter;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public SequenceToItsLocalElementLevelsConverter(ILinks<TLink> links,
            ↪  IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter) : base(links)
            ↪  => _linkToItsFrequencyToNumberConveter = linkToItsFrequencyToNumberConveter;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public IList<TLink> Convert(IList<TLink> sequence)
20         {
21             var levels = new TLink[sequence.Count];
22             levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
23             for (var i = 1; i < sequence.Count - 1; i++)
```

```
24              {
25                  var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
26                  var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
27                  levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
28              }
29              levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
   ↪  sequence[sequence.Count - 1]);
30              return levels;
31          }
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          public TLink GetFrequencyNumber(TLink source, TLink target) =>
   ↪  _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLink>(source, target));
35      }
36  }
```

## 1.125 ./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/DefaultSequenceElementCriterionMatcher.c

```
1   using System.Runtime.CompilerServices;
2   using Platform.Interfaces;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Sequences.CriterionMatchers
7   {
8       public class DefaultSequenceElementCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
   ↪  ICriterionMatcher<TLink>
9       {
10          [MethodImpl(MethodImplOptions.AggressiveInlining)]
11          public DefaultSequenceElementCriterionMatcher(ILinks<TLink> links) : base(links) { }
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          public bool IsMatched(TLink argument) => _links.IsPartialPoint(argument);
15      }
16  }
```

## 1.126 ./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/MarkedSequenceCriterionMatcher.cs

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Interfaces;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.Sequences.CriterionMatchers
8   {
9       public class MarkedSequenceCriterionMatcher<TLink> : ICriterionMatcher<TLink>
10      {
11          private static readonly EqualityComparer<TLink> _equalityComparer =
   ↪  EqualityComparer<TLink>.Default;
12
13          private readonly ILinks<TLink> _links;
14          private readonly TLink _sequenceMarkerLink;
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          public MarkedSequenceCriterionMatcher(ILinks<TLink> links, TLink sequenceMarkerLink)
18          {
19              _links = links;
20              _sequenceMarkerLink = sequenceMarkerLink;
21          }
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          public bool IsMatched(TLink sequenceCandidate)
25              => _equalityComparer.Equals(_links.GetSource(sequenceCandidate), _sequenceMarkerLink)
26              || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarkerLink,
   ↪  sequenceCandidate), _links.Constants.Null);
27      }
28  }
```

## 1.127 ./csharp/Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Collections.Stacks;
4   using Platform.Data.Doublets.Sequences.HeightProviders;
5   using Platform.Data.Sequences;
6
7   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9   namespace Platform.Data.Doublets.Sequences
10  {
11      public class DefaultSequenceAppender<TLink> : LinksOperatorBase<TLink>,
   ↪  ISequenceAppender<TLink>
```

```
12      {
13          private static readonly EqualityComparer<TLink> _equalityComparer =
              ↪  EqualityComparer<TLink>.Default;
14
15          private readonly IStack<TLink> _stack;
16          private readonly ISequenceHeightProvider<TLink> _heightProvider;
17
18          [MethodImpl(MethodImplOptions.AggressiveInlining)]
19          public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
              ↪  ISequenceHeightProvider<TLink> heightProvider)
20              : base(links)
21          {
22              _stack = stack;
23              _heightProvider = heightProvider;
24          }
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          public TLink Append(TLink sequence, TLink appendant)
28          {
29              var cursor = sequence;
30              var links = _links;
31              while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
32              {
33                  var source = links.GetSource(cursor);
34                  var target = links.GetTarget(cursor);
35                  if (_equalityComparer.Equals(_heightProvider.Get(source),
                      ↪  _heightProvider.Get(target)))
36                  {
37                      break;
38                  }
39                  else
40                  {
41                      _stack.Push(source);
42                      cursor = target;
43                  }
44              }
45              var left = cursor;
46              var right = appendant;
47              while (!_equalityComparer.Equals(cursor = _stack.PopOrDefault(),
                  ↪  links.Constants.Null))
48              {
49                  right = links.GetOrCreate(left, right);
50                  left = cursor;
51              }
52              return links.GetOrCreate(left, right);
53          }
54      }
55  }
```

## 1.128  ./csharp/Platform.Data.Doublets/Sequences/DuplicateSegmentsCounter.cs

```
1  using System.Collections.Generic;
2  using System.Linq;
3  using System.Runtime.CompilerServices;
4  using Platform.Interfaces;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Sequences
9  {
10     public class DuplicateSegmentsCounter<TLink> : ICounter<int>
11     {
12         private readonly IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
             ↪  _duplicateFragmentsProvider;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLink>,
             ↪  IList<TLink>>>> duplicateFragmentsProvider) => _duplicateFragmentsProvider =
             ↪  duplicateFragmentsProvider;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
19     }
20  }
```

## 1.129  ./csharp/Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs

```
1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Interfaces;
```

```csharp
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()
        {
```

```csharp
                  _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)
              {
```

```csharp
                    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();
            }
        }
    }
}
```

## 1.130 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequenciesCache.cs

```csharp
using System;
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.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;
        }
```

```csharp
        [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?
```

```csharp
112                    if (((_comparer.Compare(frequency, count) > 0) &&
                        ↪  (_comparer.Compare(Arithmetic.Subtract(frequency, count), _one) > 0))
113                        || ((_comparer.Compare(count, frequency) > 0) &&
                        ↪  (_comparer.Compare(Arithmetic.Subtract(count, frequency), _one) > 0)))
114                    {
115                        throw new InvalidOperationException("Frequencies validation failed.");
116                    }
117                }
118                //else
119                //{
120                //    if (value.Frequency > 0)
121                //    {
122                //        var frequency = value.Frequency;
123                //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
124                //        var count = _countLinkFrequency(linkIndex);
125
126                //        if ((frequency > count && frequency - count > 1) || (count > frequency
                        ↪  && count - frequency > 1))
127                //            throw new InvalidOperationException("Frequencies validation
                        ↪  failed.");
128                //    }
129                //}
130            }
131        }
132    }
133 }
```

## 1.131   ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs

```csharp
1  using System.Runtime.CompilerServices;
2  using Platform.Numbers;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
7  {
8      public class LinkFrequency<TLink>
9      {
10         public TLink Frequency { get; set; }
11         public TLink Link { get; set; }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public LinkFrequency(TLink frequency, TLink link)
15         {
16             Frequency = frequency;
17             Link = link;
18         }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public LinkFrequency() { }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public void IncrementFrequency() => Frequency = Arithmetic<TLink>.Increment(Frequency);
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public void DecrementFrequency() => Frequency = Arithmetic<TLink>.Decrement(Frequency);
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public override string ToString() => $"F: {Frequency}, L: {Link}";
31     }
32 }
```

## 1.132   ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToItsFrequencyValueConverter.cs

```csharp
1  using System.Runtime.CompilerServices;
2  using Platform.Converters;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
7  {
8      public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLink> :
          ↪  IConverter<Doublet<TLink>, TLink>
9      {
10         private readonly LinkFrequenciesCache<TLink> _cache;
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public
           ↪  FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink>
           ↪  cache) => _cache = cache;
14
```

```csharp
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref source).Frequency;
        }
    }
```

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

```csharp
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;
        }
```

```
41          [MethodImpl(MethodImplOptions.AggressiveInlining)]
42          private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
   ↪   _links.IsPartialPoint(x); // TODO: Use SequenceElementCreteriaMatcher instead of
   ↪   IsPartialPoint

43
44          [MethodImpl(MethodImplOptions.AggressiveInlining)]
45          private bool VisitElement(TLink element)
46          {
47              if (_equalityComparer.Equals(element, _symbol))
48              {
49                  _total = Arithmetic.Increment(_total);
50              }
51              return true;
52          }
53      }
54  }
```

## 1.135   ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequency

```
1  using System.Runtime.CompilerServices;
2  using Platform.Interfaces;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
7  {
8      public class TotalMarkedSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
9      {
10          private readonly ILinks<TLink> _links;
11          private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          public TotalMarkedSequenceSymbolFrequencyCounter(ILinks<TLink> links,
   ↪   ICriterionMatcher<TLink> markedSequenceMatcher)
15          {
16              _links = links;
17              _markedSequenceMatcher = markedSequenceMatcher;
18          }
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          public TLink Count(TLink argument) => new
   ↪   TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
   ↪   _markedSequenceMatcher, argument).Count();
22      }
23  }
```

## 1.136   ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequency

```
1  using System.Runtime.CompilerServices;
2  using Platform.Interfaces;
3  using Platform.Numbers;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
8  {
9      public class TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
   ↪   TotalSequenceSymbolFrequencyOneOffCounter<TLink>
10      {
11          private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          public TotalMarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
   ↪   ICriterionMatcher<TLink> markedSequenceMatcher, TLink symbol)
15              : base(links, symbol)
16              => _markedSequenceMatcher = markedSequenceMatcher;
17
18          [MethodImpl(MethodImplOptions.AggressiveInlining)]
19          protected override void CountSequenceSymbolFrequency(TLink link)
20          {
21              var symbolFrequencyCounter = new
   ↪   MarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
   ↪   _markedSequenceMatcher, link, _symbol);
22              _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
23          }
24      }
25  }
```

## 1.137   ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounte

```
1  using System.Runtime.CompilerServices;
2  using Platform.Interfaces;
```

```csharp
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
7   {
8       public class TotalSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
9       {
10          private readonly ILinks<TLink> _links;
11
12          [MethodImpl(MethodImplOptions.AggressiveInlining)]
13          public TotalSequenceSymbolFrequencyCounter(ILinks<TLink> links) => _links = links;
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          public TLink Count(TLink symbol) => new
            ↪   TotalSequenceSymbolFrequencyOneOffCounter<TLink>(_links, symbol).Count();
17      }
18  }
```

## 1.138 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOff

```csharp
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Interfaces;
4   using Platform.Numbers;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
9   {
10      public class TotalSequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
11      {
12          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪   EqualityComparer<TLink>.Default;
13          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
14
15          protected readonly ILinks<TLink> _links;
16          protected readonly TLink _symbol;
17          protected readonly HashSet<TLink> _visits;
18          protected TLink _total;
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
22          {
23              _links = links;
24              _symbol = symbol;
25              _visits = new HashSet<TLink>();
26              _total = default;
27          }
28
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          public TLink Count()
31          {
32              if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
33              {
34                  return _total;
35              }
36              CountCore(_symbol);
37              return _total;
38          }
39
40          [MethodImpl(MethodImplOptions.AggressiveInlining)]
41          private void CountCore(TLink link)
42          {
43              var any = _links.Constants.Any;
44              if (_equalityComparer.Equals(_links.Count(any, link), default))
45              {
46                  CountSequenceSymbolFrequency(link);
47              }
48              else
49              {
50                  _links.Each(EachElementHandler, any, link);
51              }
52          }
53
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          protected virtual void CountSequenceSymbolFrequency(TLink link)
56          {
57              var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
            ↪   link, _symbol);
58              _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
59          }
```

```csharp
            [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;
            }
        }
    }
```

## 1.139  ./csharp/Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs

```csharp
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;
        }
    }
}
```

## 1.140  ./csharp/Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs

```csharp
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
{
```

```csharp
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;
    }
}
}
```

## 1.141 ./csharp/Platform.Data.Doublets/Sequences/HeightProviders/ISequenceHeightProvider.cs

```csharp
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>
    {
    }
}
```

## 1.142 ./csharp/Platform.Data.Doublets/Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs

```csharp
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);
```

```csharp
            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);
        }
    }
}
```

## 1.143 ./csharp/Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs

```csharp
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);
            }
        }
```

```
47            return indexed;
48        }
49
50        [MethodImpl(MethodImplOptions.AggressiveInlining)]
51        private void Increment(TLink link)
52        {
53            var previousFrequency = _frequencyPropertyOperator.Get(link);
54            var frequency = _frequencyIncrementer.Increment(previousFrequency);
55            _frequencyPropertyOperator.Set(link, frequency);
56        }
57    }
58 }
```

## 1.144 ./csharp/Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.cs

```
1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences.Indexes
7 {
8     public interface ISequenceIndex<TLink>
9     {
10        /// <summary>
11        /// Индексирует последовательность глобально, и возвращает значение,
12        /// определяющие была ли запрошенная последовательность проиндексирована ранее.
13        /// </summary>
14        /// <param name="sequence">Последовательность для индексации.</param>
15        [MethodImpl(MethodImplOptions.AggressiveInlining)]
16        bool Add(IList<TLink> sequence);
17
18        [MethodImpl(MethodImplOptions.AggressiveInlining)]
19        bool MightContain(IList<TLink> sequence);
20    }
21 }
```

## 1.145 ./csharp/Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.cs

```
1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences.Indexes
7 {
8     public class SequenceIndex<TLink> : LinksOperatorBase<TLink>, ISequenceIndex<TLink>
9     {
10        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
11
12        [MethodImpl(MethodImplOptions.AggressiveInlining)]
13        public SequenceIndex(ILinks<TLink> links) : base(links) { }
14
15        [MethodImpl(MethodImplOptions.AggressiveInlining)]
16        public virtual bool Add(IList<TLink> sequence)
17        {
18            var indexed = true;
19            var i = sequence.Count;
20            while (--i >= 1 && (indexed =
            ↪  !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
            ↪  default))) { }
21            for (; i >= 1; i--)
22            {
23                _links.GetOrCreate(sequence[i - 1], sequence[i]);
24            }
25            return indexed;
26        }
27
28        [MethodImpl(MethodImplOptions.AggressiveInlining)]
29        public virtual bool MightContain(IList<TLink> sequence)
30        {
31            var indexed = true;
32            var i = sequence.Count;
33            while (--i >= 1 && (indexed =
            ↪  !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
            ↪  default))) { }
34            return indexed;
35        }
36    }
37 }
```

```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.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;
            });
        }
    }
}
```

```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.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;
    }
}
```

```csharp
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.");
```

```
77                    }
78                    variants[last++] = variant;
79                }
80            }
81        }
82        return variants;
83    }

85    [MethodImpl(MethodImplOptions.AggressiveInlining)]
86    public List<ulong> CreateAllVariants1(params ulong[] sequence)
87    {
88        return _sync.ExecuteWriteOperation(() =>
89        {
90            if (sequence.IsNullOrEmpty())
91            {
92                return new List<ulong>();
93            }
94            Links.Unsync.EnsureLinkExists(sequence);
95            if (sequence.Length == 1)
96            {
97                return new List<ulong> { sequence[0] };
98            }
99            var results = new
            ↪  List<ulong>((int)Platform.Numbers.Math.Catalan(sequence.Length));
100           return CreateAllVariants1Core(sequence, results);
101        });
102   }

104   [MethodImpl(MethodImplOptions.AggressiveInlining)]
105   private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
106   {
107       if (sequence.Length == 2)
108       {
109           var link = Links.Unsync.GetOrCreate(sequence[0], sequence[1]);
110           if (link == Constants.Null)
111           {
112               throw new NotImplementedException("Creation cancellation is not
               ↪  implemented.");
113           }
114           results.Add(link);
115           return results;
116       }
117       var innerSequenceLength = sequence.Length - 1;
118       var innerSequence = new ulong[innerSequenceLength];
119       for (var li = 0; li < innerSequenceLength; li++)
120       {
121           var link = Links.Unsync.GetOrCreate(sequence[li], sequence[li + 1]);
122           if (link == Constants.Null)
123           {
124               throw new NotImplementedException("Creation cancellation is not
               ↪  implemented.");
125           }
126           for (var isi = 0; isi < li; isi++)
127           {
128               innerSequence[isi] = sequence[isi];
129           }
130           innerSequence[li] = link;
131           for (var isi = li + 1; isi < innerSequenceLength; isi++)
132           {
133               innerSequence[isi] = sequence[isi + 1];
134           }
135           CreateAllVariants1Core(innerSequence, results);
136       }
137       return results;
138   }

140   #endregion

142   [MethodImpl(MethodImplOptions.AggressiveInlining)]
143   public HashSet<ulong> Each1(params ulong[] sequence)
144   {
145       var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
146       Each1(link =>
147       {
148           if (!visitedLinks.Contains(link))
149           {
150               visitedLinks.Add(link); // изучить почему случаются повторы
151           }
```

```csharp
                        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);
            }
```

```csharp
            [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;
                });
```

```csharp
        }

        [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)
```

```csharp
                    {
                        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;
            });
        }
```

```csharp
        [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))
                        //{
```

```
531             //    sb.Append('{');
532             //    elementToString(sb, element);
533             //    sb.Append('}');
534             //}
535             //else
536                 elementToString(sb, element);
537                 if (sb.Length < MaxSequenceFormatSize)
538                 {
539                     return true;
540                 }
541                 sb.Append(insertComma ? ", ..." : "...");
542                 return false;
543             });
544         }
545         sb.Append('}');
546         return sb.ToString();
547     }
548
549     [MethodImpl(MethodImplOptions.AggressiveInlining)]
550     public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
    ↪  knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
    ↪  knownElements);
551
552     [MethodImpl(MethodImplOptions.AggressiveInlining)]
553     public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
    ↪  LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
    ↪  Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
    ↪  sequenceLink, elementToString, insertComma, knownElements));
554
555     [MethodImpl(MethodImplOptions.AggressiveInlining)]
556     private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
    ↪  Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
    ↪  LinkIndex[] knownElements)
557     {
558         var linksInSequence = new HashSet<ulong>(knownElements);
559         var entered = new HashSet<ulong>();
560         var sb = new StringBuilder();
561         sb.Append('{');
562         if (links.Exists(sequenceLink))
563         {
564             StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
565                 x => linksInSequence.Contains(x) || links.IsFullPoint(x),
    ↪  entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
566             {
567                 if (insertComma && sb.Length > 1)
568                 {
569                     sb.Append(',');
570                 }
571                 if (entered.Contains(element))
572                 {
573                     sb.Append('{');
574                     elementToString(sb, element);
575                     sb.Append('}');
576                 }
577                 else
578                 {
579                     elementToString(sb, element);
580                 }
581                 if (sb.Length < MaxSequenceFormatSize)
582                 {
583                     return true;
584                 }
585                 sb.Append(insertComma ? ", ..." : "...");
586                 return false;
587             });
588         }
589         sb.Append('}');
590         return sb.ToString();
591     }
592
593     [MethodImpl(MethodImplOptions.AggressiveInlining)]
594     public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
595     {
596         return _sync.ExecuteReadOperation(() =>
597         {
598             if (sequence.Length > 0)
599             {
600                 Links.EnsureLinkExists(sequence);
```

```csharp
                        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(() =>
            {
```

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

```csharp
754          [MethodImpl(MethodImplOptions.AggressiveInlining)]
755          public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
          ↪  IList<ulong> sequence)
756          {
757              return _sync.ExecuteReadOperation(() =>
758              {
759                  if (sequence.Count > 0)
760                  {
761                      Links.EnsureLinkExists(sequence);
762                      var results = new HashSet<LinkIndex>();
763                      //var nextResults = new HashSet<ulong>();
764                      //for (var i = 0; i < sequence.Length; i++)
765                      //{
766                      //    AllUsagesCore(sequence[i], nextResults);
767                      //    if (results.IsNullOrEmpty())
768                      //    {
769                      //        results = nextResults;
770                      //        nextResults = new HashSet<ulong>();
771                      //    }
772                      //    else
773                      //    {
774                      //        results.IntersectWith(nextResults);
775                      //        nextResults.Clear();
776                      //    }
777                      //}
778                      var collector1 = new AllUsagesCollector1(Links.Unsync, results);
779                      collector1.Collect(Links.Unsync.GetLink(sequence[0]));
780                      var next = new HashSet<ulong>();
781                      for (var i = 1; i < sequence.Count; i++)
782                      {
783                          var collector = new AllUsagesCollector1(Links.Unsync, next);
784                          collector.Collect(Links.Unsync.GetLink(sequence[i]));

786                          results.IntersectWith(next);
787                          next.Clear();
788                      }
789                      var filteredResults = new HashSet<ulong>();
790                      var matcher = new Matcher(this, sequence, filteredResults, null,
                      ↪  readAsElements);
791                      matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
                      ↪  x)); // OrderBy is a Hack
792                      return filteredResults;
793                  }
794                  return new HashSet<ulong>();
795              });
796          }

798          // Does not work
799          //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
          ↪  params ulong[] sequence)
800          //{
801          //    var visited = new HashSet<ulong>();
802          //    var results = new HashSet<ulong>();
803          //    var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
          ↪  true; }, readAsElements);
804          //    var last = sequence.Length - 1;
805          //    for (var i = 0; i < last; i++)
806          //    {
807          //        PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
808          //    }
809          //    return results;
810          //}

812          [MethodImpl(MethodImplOptions.AggressiveInlining)]
813          public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
814          {
815              return _sync.ExecuteReadOperation(() =>
816              {
817                  if (sequence.Length > 0)
818                  {
819                      Links.EnsureLinkExists(sequence);
820                      //var firstElement = sequence[0];
821                      //if (sequence.Length == 1)
822                      //{
823                      //    //results.Add(firstElement);
824                      //    return results;
825                      //}
```

```csharp
            //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>();
```

```
897                         foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
                            ↪   firstLinkUsages, 1))
898                         {
899                             AllUsagesCore(match, results);
900                         }
901                         return results.ToList();
902                     }
903                     return new List<ulong>();
904                 });
905         }
906
907         /// <remarks>
908         /// TODO: Может потребоваться ограничение на уровень глубины рекурсии
909         /// </remarks>
910         [MethodImpl(MethodImplOptions.AggressiveInlining)]
911         public HashSet<ulong> AllUsages(ulong link)
912         {
913             return _sync.ExecuteReadOperation(() =>
914             {
915                 var usages = new HashSet<ulong>();
916                 AllUsagesCore(link, usages);
917                 return usages;
918             });
919         }
920
921         // При сборе всех использований (последовательностей) можно сохранять обратный путь к
                ↪   той связи с которой начинался поиск (STTTSSSTT),
922         // причём достаточно одного бита для хранения перехода влево или вправо
923         [MethodImpl(MethodImplOptions.AggressiveInlining)]
924         private void AllUsagesCore(ulong link, HashSet<ulong> usages)
925         {
926             bool handler(ulong doublet)
927             {
928                 if (usages.Add(doublet))
929                 {
930                     AllUsagesCore(doublet, usages);
931                 }
932                 return true;
933             }
934             Links.Unsync.Each(link, Constants.Any, handler);
935             Links.Unsync.Each(Constants.Any, link, handler);
936         }
937
938         [MethodImpl(MethodImplOptions.AggressiveInlining)]
939         public HashSet<ulong> AllBottomUsages(ulong link)
940         {
941             return _sync.ExecuteReadOperation(() =>
942             {
943                 var visits = new HashSet<ulong>();
944                 var usages = new HashSet<ulong>();
945                 AllBottomUsagesCore(link, visits, usages);
946                 return usages;
947             });
948         }
949
950         [MethodImpl(MethodImplOptions.AggressiveInlining)]
951         private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
                ↪   usages)
952         {
953             bool handler(ulong doublet)
954             {
955                 if (visits.Add(doublet))
956                 {
957                     AllBottomUsagesCore(doublet, visits, usages);
958                 }
959                 return true;
960             }
961             if (Links.Unsync.Count(Constants.Any, link) == 0)
962             {
963                 usages.Add(link);
964             }
965             else
966             {
967                 Links.Unsync.Each(link, Constants.Any, handler);
968                 Links.Unsync.Each(Constants.Any, link, handler);
969             }
970         }
971
```

```csharp
        [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)
```

```csharp
                    {
                        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);
```

```csharp
                        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;
```

```csharp
            [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);
```

```csharp
                    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)
                        {
```

```csharp
                            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);
```

```csharp
                        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)
                {
```

```csharp
                    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);
                        }
                    }
                }
```

```csharp
                    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>
```

```csharp
    {
        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;
```

```csharp
                            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)
```

```csharp
            //{
            //    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)
```

```csharp
                                {
                                    _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);
                        }
                    }
                }
            }
```

```
2039              #endregion
2040          }
2041  }

1.149    ./csharp/Platform.Data.Doublets/Sequences/Sequences.cs
   1  using System;
   2  using System.Collections.Generic;
   3  using System.Linq;
   4  using System.Runtime.CompilerServices;
   5  using Platform.Collections;
   6  using Platform.Collections.Lists;
   7  using Platform.Collections.Stacks;
   8  using Platform.Threading.Synchronization;
   9  using Platform.Data.Doublets.Sequences.Walkers;
  10  using LinkIndex = System.UInt64;
  11
  12  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
  13
  14  namespace Platform.Data.Doublets.Sequences
  15  {
  16      /// <summary>
  17      /// Представляет коллекцию последовательностей связей.
  18      /// </summary>
  19      /// <remarks>
  20      /// Обязательно реализовать атомарность каждого публичного метода.
  21      ///
  22      /// TODO:
  23      ///
  24      /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
  25      /// через естественную группировку по unicode типам, все whitespace вместе, все символы
  26      ///     вместе, все числа вместе и т.п.
  27      /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
  28      ///     графа)
  29      ///
  30      /// x*y - найти все связи между, в последовательностях любой формы, если не стоит
  31      ///     ограничитель на то, что является последовательностью, а что нет,
  32      /// то находятся любые структуры связей, которые содержат эти элементы именно в таком
  33      ///     порядке.
  34      ///
  35      /// Рост последовательности слева и справа.
  36      /// Поиск со звёздочкой.
  37      /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
  38      /// так же проблема может быть решена при реализации дистанционных триггеров.
  39      /// Нужны ли уникальные указатели вообще?
  40      /// Что если обращение к информации будет происходить через содержимое всегда?
  41      ///
  42      /// Писать тесты.
  43      ///
  44      ///
  45      /// Можно убрать зависимость от конкретной реализации Links,
  46      /// на зависимость от абстрактного элемента, который может быть представлен несколькими
  47      ///     способами.
  48      ///
  49      /// Можно ли как-то сделать один общий интерфейс
  50      ///
  51      ///
  52      /// Блокчейн и/или гит для распределённой записи транзакций.
  53      ///
  54      /// </remarks>
  55      public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
  56          (после завершения реализации Sequences)
  57      {
  58          /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
  59          ///     связей.</summary>
  60          public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;
  61
  62          public SequencesOptions<LinkIndex> Options { get; }
  63          public SynchronizedLinks<LinkIndex> Links { get; }
  64          private readonly ISynchronization _sync;
  65
  66          public LinksConstants<LinkIndex> Constants { get; }
  67
  68          [MethodImpl(MethodImplOptions.AggressiveInlining)]
  69          public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
  70          {
  71              Links = links;
  72              _sync = links.SyncRoot;
  73              Options = options;
  74              Options.ValidateOptions();
```

```csharp
                    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)]
```

```csharp
146        private LinkIndex CountUsages(params LinkIndex[] restrictions)
147        {
148            if (restrictions.Length == 0)
149            {
150                return 0;
151            }
152            if (restrictions.Length == 1) // Первая связь это адрес
153            {
154                if (restrictions[0] == Constants.Null)
155                {
156                    return 0;
157                }
158                var any = Constants.Any;
159                if (Options.UseSequenceMarker)
160                {
161                    var elementsLink = GetSequenceElements(restrictions[0]);
162                    var sequenceLink = GetSequenceByElements(elementsLink);
163                    if (sequenceLink != Constants.Null)
164                    {
165                        return Links.Count(any, sequenceLink) + Links.Count(any, elementsLink) -
   ↪    1;
166                    }
167                    return Links.Count(any, elementsLink);
168                }
169                return Links.Count(any, restrictions[0]);
170            }
171            throw new NotImplementedException();
172        }
173
174        #endregion
175
176        #region Create
177
178        [MethodImpl(MethodImplOptions.AggressiveInlining)]
179        public LinkIndex Create(IList<LinkIndex> restrictions)
180        {
181            return _sync.ExecuteWriteOperation(() =>
182            {
183                if (restrictions.IsNullOrEmpty())
184                {
185                    return Constants.Null;
186                }
187                Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
188                return CreateCore(restrictions);
189            });
190        }
191
192        [MethodImpl(MethodImplOptions.AggressiveInlining)]
193        private LinkIndex CreateCore(IList<LinkIndex> restrictions)
194        {
195            LinkIndex[] sequence = restrictions.SkipFirst();
196            if (Options.UseIndex)
197            {
198                Options.Index.Add(sequence);
199            }
200            var sequenceRoot = default(LinkIndex);
201            if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
202            {
203                var matches = Each(restrictions);
204                if (matches.Count > 0)
205                {
206                    sequenceRoot = matches[0];
207                }
208            }
209            else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
210            {
211                return CompactCore(sequence);
212            }
213            if (sequenceRoot == default)
214            {
215                sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
216            }
217            if (Options.UseSequenceMarker)
218            {
219                return Links.Unsync.GetOrCreate(Options.SequenceMarkerLink, sequenceRoot);
220            }
221            return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
222        }
```

```csharp
223
224          #endregion
225
226          #region Each
227
228          [MethodImpl(MethodImplOptions.AggressiveInlining)]
229          public List<LinkIndex> Each(IList<LinkIndex> sequence)
230          {
231              var results = new List<LinkIndex>();
232              var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
233              Each(filler.AddFirstAndReturnConstant, sequence);
234              return results;
235          }
236
237          [MethodImpl(MethodImplOptions.AggressiveInlining)]
238          public LinkIndex Each(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
      ↪    restrictions)
239          {
240              return _sync.ExecuteReadOperation(() =>
241              {
242                  if (restrictions.IsNullOrEmpty())
243                  {
244                      return Constants.Continue;
245                  }
246                  Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
247                  if (restrictions.Count == 1)
248                  {
249                      var link = restrictions[0];
250                      var any = Constants.Any;
251                      if (link == any)
252                      {
253                          if (Options.UseSequenceMarker)
254                          {
255                              return Links.Unsync.Each(handler, new Link<LinkIndex>(any,
      ↪            Options.SequenceMarkerLink, any));
256                          }
257                          else
258                          {
259                              return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
      ↪            any));
260                          }
261                      }
262                      if (Options.UseSequenceMarker)
263                      {
264                          var sequenceLinkValues = Links.Unsync.GetLink(link);
265                          if (sequenceLinkValues[Constants.SourcePart] ==
      ↪            Options.SequenceMarkerLink)
266                          {
267                              link = sequenceLinkValues[Constants.TargetPart];
268                          }
269                      }
270                      var sequence = Options.Walker.Walk(link).ToArray().ShiftRight();
271                      sequence[0] = link;
272                      return handler(sequence);
273                  }
274                  else if (restrictions.Count == 2)
275                  {
276                      throw new NotImplementedException();
277                  }
278                  else if (restrictions.Count == 3)
279                  {
280                      return Links.Unsync.Each(handler, restrictions);
281                  }
282                  else
283                  {
284                      var sequence = restrictions.SkipFirst();
285                      if (Options.UseIndex && !Options.Index.MightContain(sequence))
286                      {
287                          return Constants.Break;
288                      }
289                      return EachCore(handler, sequence);
290                  }
291              });
292          }
293
294          [MethodImpl(MethodImplOptions.AggressiveInlining)]
295          private LinkIndex EachCore(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
      ↪    values)
```

```csharp
            {
                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)]
```

```csharp
362        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));
363
364        [MethodImpl(MethodImplOptions.AggressiveInlining)]
365        private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
      ↪    left, LinkIndex stepFrom)
366        {
367            var upStep = stepFrom;
368            var firstTarget = Links.Unsync.GetSource(upStep);
369            while (firstTarget != left && firstTarget != upStep)
370            {
371                upStep = firstTarget;
372                firstTarget = Links.Unsync.GetTarget(upStep);
373            }
374            if (firstTarget == left)
375            {
376                return handler(new LinkAddress<LinkIndex>(stepFrom));
377            }
378            return Constants.Continue;
379        }
380
381        #endregion
382
383        #region Update
384
385        [MethodImpl(MethodImplOptions.AggressiveInlining)]
386        public LinkIndex Update(IList<LinkIndex> restrictions, IList<LinkIndex> substitution)
387        {
388            var sequence = restrictions.SkipFirst();
389            var newSequence = substitution.SkipFirst();
390            if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
391            {
392                return Constants.Null;
393            }
394            if (sequence.IsNullOrEmpty())
395            {
396                return Create(substitution);
397            }
398            if (newSequence.IsNullOrEmpty())
399            {
400                Delete(restrictions);
401                return Constants.Null;
402            }
403            return _sync.ExecuteWriteOperation((Func<ulong>)(() =>
404            {
405                ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links, (IList<ulong>)sequence);
406                Links.EnsureLinkExists(newSequence);
407                return UpdateCore(sequence, newSequence);
408            }));
409        }
410
411        [MethodImpl(MethodImplOptions.AggressiveInlining)]
412        private LinkIndex UpdateCore(IList<LinkIndex> sequence, IList<LinkIndex> newSequence)
413        {
414            LinkIndex bestVariant;
415            if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
      ↪    !sequence.EqualTo(newSequence))
416            {
417                bestVariant = CompactCore(newSequence);
418            }
419            else
420            {
421                bestVariant = CreateCore(newSequence);
422            }
423            // TODO: Check all options only ones before loop execution
424            // Возможно нужно две версии Each, возвращающий фактические последовательности и с
      ↪    маркером,
425            // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
      ↪    можно получить имея только фактические последовательности.
426            foreach (var variant in Each(sequence))
427            {
428                if (variant != bestVariant)
429                {
430                    UpdateOneCore(variant, bestVariant);
431                }
432            }
433            return bestVariant;
```

```csharp
434                }
435
436            [MethodImpl(MethodImplOptions.AggressiveInlining)]
437            private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
438            {
439                if (Options.UseGarbageCollection)
440                {
441                    var sequenceElements = GetSequenceElements(sequence);
442                    var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
443                    var sequenceLink = GetSequenceByElements(sequenceElements);
444                    var newSequenceElements = GetSequenceElements(newSequence);
445                    var newSequenceLink = GetSequenceByElements(newSequenceElements);
446                    if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
447                    {
448                        if (sequenceLink != Constants.Null)
449                        {
450                            Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
451                        }
452                        Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
453                    }
454                    ClearGarbage(sequenceElementsContents.Source);
455                    ClearGarbage(sequenceElementsContents.Target);
456                }
457                else
458                {
459                    if (Options.UseSequenceMarker)
460                    {
461                        var sequenceElements = GetSequenceElements(sequence);
462                        var sequenceLink = GetSequenceByElements(sequenceElements);
463                        var newSequenceElements = GetSequenceElements(newSequence);
464                        var newSequenceLink = GetSequenceByElements(newSequenceElements);
465                        if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
466                        {
467                            if (sequenceLink != Constants.Null)
468                            {
469                                Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
470                            }
471                            Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
472                        }
473                    }
474                    else
475                    {
476                        if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
477                        {
478                            Links.Unsync.MergeAndDelete(sequence, newSequence);
479                        }
480                    }
481                }
482            }
483
484            #endregion
485
486            #region Delete
487
488            [MethodImpl(MethodImplOptions.AggressiveInlining)]
489            public void Delete(IList<LinkIndex> restrictions)
490            {
491                _sync.ExecuteWriteOperation(() =>
492                {
493                    var sequence = restrictions.SkipFirst();
494                    // TODO: Check all options only ones before loop execution
495                    foreach (var linkToDelete in Each(sequence))
496                    {
497                        DeleteOneCore(linkToDelete);
498                    }
499                });
500            }
501
502            [MethodImpl(MethodImplOptions.AggressiveInlining)]
503            private void DeleteOneCore(LinkIndex link)
504            {
505                if (Options.UseGarbageCollection)
506                {
507                    var sequenceElements = GetSequenceElements(link);
508                    var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
509                    var sequenceLink = GetSequenceByElements(sequenceElements);
510                    if (Options.UseCascadeDelete || CountUsages(link) == 0)
511                    {
```

```csharp
                            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
```

```csharp
        #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))
```

```csharp
663                    {
664                        break;
665                    }
666                }
667                return _filterPosition == _patternSequence.Count;
668            }
669
670            [MethodImpl(MethodImplOptions.AggressiveInlining)]
671            private bool FullMatchCore(LinkIndex element)
672            {
673                if (_filterPosition == _patternSequence.Count)
674                {
675                    _filterPosition = -2; // Длиннее чем нужно
676                    return false;
677                }
678                if (_patternSequence[_filterPosition] != _links.Constants.Any
679                 && element != _patternSequence[_filterPosition])
680                {
681                    _filterPosition = -1;
682                    return false; // Начинается/Продолжается иначе
683                }
684                _filterPosition++;
685                return true;
686            }
687
688            [MethodImpl(MethodImplOptions.AggressiveInlining)]
689            public void AddFullMatchedToResults(IList<LinkIndex> restrictions)
690            {
691                var sequenceToMatch = restrictions[_links.Constants.IndexPart];
692                if (FullMatch(sequenceToMatch))
693                {
694                    _results.Add(sequenceToMatch);
695                }
696            }
697
698            [MethodImpl(MethodImplOptions.AggressiveInlining)]
699            public LinkIndex HandleFullMatched(IList<LinkIndex> restrictions)
700            {
701                var sequenceToMatch = restrictions[_links.Constants.IndexPart];
702                if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
703                {
704                    return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
705                }
706                return _links.Constants.Continue;
707            }
708
709            [MethodImpl(MethodImplOptions.AggressiveInlining)]
710            public LinkIndex HandleFullMatchedSequence(IList<LinkIndex> restrictions)
711            {
712                var sequenceToMatch = restrictions[_links.Constants.IndexPart];
713                var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
714                if (sequence != _links.Constants.Null && FullMatch(sequenceToMatch) &&
                    ↪ _results.Add(sequenceToMatch))
715                {
716                    return _stopableHandler(new LinkAddress<LinkIndex>(sequence));
717                }
718                return _links.Constants.Continue;
719            }
720
721            /// <remarks>
722            /// TODO: Add support for LinksConstants.Any
723            /// </remarks>
724            [MethodImpl(MethodImplOptions.AggressiveInlining)]
725            public bool PartialMatch(LinkIndex sequenceToMatch)
726            {
727                _filterPosition = -1;
728                foreach (var part in Walk(sequenceToMatch))
729                {
730                    if (!PartialMatchCore(part))
731                    {
732                        break;
733                    }
734                }
735                return _filterPosition == _patternSequence.Count - 1;
736            }
737
738            [MethodImpl(MethodImplOptions.AggressiveInlining)]
739            private bool PartialMatchCore(LinkIndex element)
740            {
```

```csharp
                    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
    }
}
```

## 1.150   ./csharp/Platform.Data.Doublets/Sequences/SequencesExtensions.cs

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

```csharp
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;
        }
    }
}
```

## 1.151 ./csharp/Platform.Data.Doublets/Sequences/SequencesOptions.cs

```csharp
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;
```

```csharp
        }

        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;
        }
```

```csharp
        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)
            {
```

```
196                Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
197            }
198        }
199
200        [MethodImpl(MethodImplOptions.AggressiveInlining)]
201        public void ValidateOptions()
202        {
203            if (UseGarbageCollection && !UseSequenceMarker)
204            {
205                throw new NotSupportedException("To use garbage collection UseSequenceMarker
       ↪   option must be on.");
206            }
207        }
208    }
209 }
```

## 1.152 ./csharp/Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.cs

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.Walkers
7  {
8      public interface ISequenceWalker<TLink>
9      {
10          [MethodImpl(MethodImplOptions.AggressiveInlining)]
11          IEnumerable<TLink> Walk(TLink sequence);
12      }
13 }
```

## 1.153 ./csharp/Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Collections.Stacks;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Sequences.Walkers
9  {
10     public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
11     {
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
      ↪   isElement) : base(links, stack, isElement) { }
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links, stack,
      ↪   links.IsPartialPoint) { }
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override TLink GetNextElementAfterPop(TLink element) =>
      ↪   _links.GetSource(element);
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected override TLink GetNextElementAfterPush(TLink element) =>
      ↪   _links.GetTarget(element);
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override IEnumerable<TLink> WalkContents(TLink element)
26         {
27             var links = _links;
28             var parts = links.GetLink(element);
29             var start = links.Constants.SourcePart;
30             for (var i = parts.Count - 1; i >= start; i--)
31             {
32                 var part = parts[i];
33                 if (IsElement(part))
34                 {
35                     yield return part;
36                 }
37             }
38         }
39     }
40 }
```

```csharp
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  //#define USEARRAYPOOL
8  #if USEARRAYPOOL
9  using Platform.Collections;
10 #endif
11
12 namespace Platform.Data.Doublets.Sequences.Walkers
13 {
14     public class LeveledSequenceWalker<TLink> : LinksOperatorBase<TLink>, ISequenceWalker<TLink>
15     {
16         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
17
18         private readonly Func<TLink, bool> _isElement;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public LeveledSequenceWalker(ILinks<TLink> links, Func<TLink, bool> isElement) :
            ↪  base(links) => _isElement = isElement;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public LeveledSequenceWalker(ILinks<TLink> links) : base(links) => _isElement =
            ↪  _links.IsPartialPoint;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public IEnumerable<TLink> Walk(TLink sequence) => ToArray(sequence);
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public TLink[] ToArray(TLink sequence)
31         {
32             var length = 1;
33             var array = new TLink[length];
34             array[0] = sequence;
35             if (_isElement(sequence))
36             {
37                 return array;
38             }
39             bool hasElements;
40             do
41             {
42                 length *= 2;
43 #if USEARRAYPOOL
44                 var nextArray = ArrayPool.Allocate<ulong>(length);
45 #else
46                 var nextArray = new TLink[length];
47 #endif
48                 hasElements = false;
49                 for (var i = 0; i < array.Length; i++)
50                 {
51                     var candidate = array[i];
52                     if (_equalityComparer.Equals(array[i], default))
53                     {
54                         continue;
55                     }
56                     var doubletOffset = i * 2;
57                     if (_isElement(candidate))
58                     {
59                         nextArray[doubletOffset] = candidate;
60                     }
61                     else
62                     {
63                         var links = _links;
64                         var link = links.GetLink(candidate);
65                         var linkSource = links.GetSource(link);
66                         var linkTarget = links.GetTarget(link);
67                         nextArray[doubletOffset] = linkSource;
68                         nextArray[doubletOffset + 1] = linkTarget;
69                         if (!hasElements)
70                         {
71                             hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
72                         }
73                     }
74                 }
75 #if USEARRAYPOOL
76                 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;
        }
    }
}
```

## 1.155 ./csharp/Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs

```csharp
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);
```

```
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          protected override IEnumerable<TLink> WalkContents(TLink element)
26          {
27              var parts = _links.GetLink(element);
28              for (var i = _links.Constants.SourcePart; i < parts.Count; i++)
29              {
30                  var part = parts[i];
31                  if (IsElement(part))
32                  {
33                      yield return part;
34                  }
35              }
36          }
37      }
38  }
```

## 1.156  ./csharp/Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using Platform.Collections.Stacks;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Data.Doublets.Sequences.Walkers
9   {
10      public abstract class SequenceWalkerBase<TLink> : LinksOperatorBase<TLink>,
        ↪  ISequenceWalker<TLink>
11      {
12          private readonly IStack<TLink> _stack;
13          private readonly Func<TLink, bool> _isElement;
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
        ↪  isElement) : base(links)
17          {
18              _stack = stack;
19              _isElement = isElement;
20          }
21
22          [MethodImpl(MethodImplOptions.AggressiveInlining)]
23          protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : this(links,
        ↪  stack, links.IsPartialPoint) { }
24
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          public IEnumerable<TLink> Walk(TLink sequence)
27          {
28              _stack.Clear();
29              var element = sequence;
30              if (IsElement(element))
31              {
32                  yield return element;
33              }
34              else
35              {
36                  while (true)
37                  {
38                      if (IsElement(element))
39                      {
40                          if (_stack.IsEmpty)
41                          {
42                              break;
43                          }
44                          element = _stack.Pop();
45                          foreach (var output in WalkContents(element))
46                          {
47                              yield return output;
48                          }
49                          element = GetNextElementAfterPop(element);
50                      }
51                      else
52                      {
53                          _stack.Push(element);
54                          element = GetNextElementAfterPush(element);
55                      }
56                  }
57              }
58          }
59
```

```
60          [MethodImpl(MethodImplOptions.AggressiveInlining)]
61          protected virtual bool IsElement(TLink elementLink) => _isElement(elementLink);
62
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          protected abstract TLink GetNextElementAfterPop(TLink element);
65
66          [MethodImpl(MethodImplOptions.AggressiveInlining)]
67          protected abstract TLink GetNextElementAfterPush(TLink element);
68
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          protected abstract IEnumerable<TLink> WalkContents(TLink element);
71      }
72  }
```

## 1.157 ./csharp/Platform.Data.Doublets/Stacks/Stack.cs

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Collections.Stacks;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.Stacks
8   {
9       public class Stack<TLink> : LinksOperatorBase<TLink>, IStack<TLink>
10      {
11          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪   EqualityComparer<TLink>.Default;
12
13          private readonly TLink _stack;
14
15          public bool IsEmpty
16          {
17              [MethodImpl(MethodImplOptions.AggressiveInlining)]
18              get => _equalityComparer.Equals(Peek(), _stack);
19          }
20
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          public Stack(ILinks<TLink> links, TLink stack) : base(links) => _stack = stack;
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          private TLink GetStackMarker() => _links.GetSource(_stack);
26
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          private TLink GetTop() => _links.GetTarget(_stack);
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public TLink Peek() => _links.GetTarget(GetTop());
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          public TLink Pop()
35          {
36              var element = Peek();
37              if (!_equalityComparer.Equals(element, _stack))
38              {
39                  var top = GetTop();
40                  var previousTop = _links.GetSource(top);
41                  _links.Update(_stack, GetStackMarker(), previousTop);
42                  _links.Delete(top);
43              }
44              return element;
45          }
46
47          [MethodImpl(MethodImplOptions.AggressiveInlining)]
48          public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
            ↪   _links.GetOrCreate(GetTop(), element));
49      }
50  }
```

## 1.158 ./csharp/Platform.Data.Doublets/Stacks/StackExtensions.cs

```
1   using System.Runtime.CompilerServices;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Data.Doublets.Stacks
6   {
7       public static class StackExtensions
8       {
9           [MethodImpl(MethodImplOptions.AggressiveInlining)]
10          public static TLink CreateStack<TLink>(this ILinks<TLink> links, TLink stackMarker)
11          {
```

```
12          var stackPoint = links.CreatePoint();
13          var stack = links.Update(stackPoint, stackMarker, stackPoint);
14          return stack;
15        }
16      }
17    }
```

## 1.159 ./csharp/Platform.Data.Doublets/SynchronizedLinks.cs

```csharp
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Data.Doublets;
5  using Platform.Threading.Synchronization;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets
10 {
11     /// <remarks>
12     /// TODO: Autogeneration of synchronized wrapper (decorator).
13     /// TODO: Try to unfold code of each method using IL generation for performance improvements.
14     /// TODO: Or even to unfold multiple layers of implementations.
15     /// </remarks>
16     public class SynchronizedLinks<TLinkAddress> : ISynchronizedLinks<TLinkAddress>
17     {
18         public LinksConstants<TLinkAddress> Constants
19         {
20             [MethodImpl(MethodImplOptions.AggressiveInlining)]
21             get;
22         }
23
24         public ISynchronization SyncRoot
25         {
26             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             get;
28         }
29
30         public ILinks<TLinkAddress> Sync
31         {
32             [MethodImpl(MethodImplOptions.AggressiveInlining)]
33             get;
34         }
35
36         public ILinks<TLinkAddress> Unsync
37         {
38             [MethodImpl(MethodImplOptions.AggressiveInlining)]
39             get;
40         }
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         public SynchronizedLinks(ILinks<TLinkAddress> links) : this(new
↪            ReaderWriterLockSynchronization(), links) { }
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         public SynchronizedLinks(ISynchronization synchronization, ILinks<TLinkAddress> links)
47         {
48             SyncRoot = synchronization;
49             Sync = this;
50             Unsync = links;
51             Constants = links.Constants;
52         }
53
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]
55         public TLinkAddress Count(IList<TLinkAddress> restriction) =>
↪            SyncRoot.ExecuteReadOperation(restriction, Unsync.Count);
56
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         public TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler,
↪            IList<TLinkAddress> restrictions) => SyncRoot.ExecuteReadOperation(handler,
↪            restrictions, (handler1, restrictions1) => Unsync.Each(handler1, restrictions1));
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         public TLinkAddress Create(IList<TLinkAddress> restrictions) =>
↪            SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Create);
62
63         [MethodImpl(MethodImplOptions.AggressiveInlining)]
64         public TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress>
↪            substitution) => SyncRoot.ExecuteWriteOperation(restrictions, substitution,
↪            Unsync.Update);
65
```

```
66          [MethodImpl(MethodImplOptions.AggressiveInlining)]
67          public void Delete(IList<TLinkAddress> restrictions) =>
            ↪   SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Delete);
68
69          //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
            ↪   IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
70          //{
71          //    if (restriction != null && substitution != null &&
            ↪   !substitution.EqualTo(restriction))
72          //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
            ↪   substitution, substitutedHandler, Unsync.Trigger);
73
74          //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
            ↪   substitutedHandler, Unsync.Trigger);
75          //}
76      }
77  }
```

## 1.160  ./csharp/Platform.Data.Doublets/Time/DateTimeToLongRawNumberSequenceConverter.cs

```
1   using System;
2   using System.Runtime.CompilerServices;
3   using Platform.Converters;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.Time
8   {
9       public class DateTimeToLongRawNumberSequenceConverter<TLink> : IConverter<DateTime, TLink>
10      {
11          private readonly IConverter<long, TLink> _int64ToLongRawNumberConverter;
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          public DateTimeToLongRawNumberSequenceConverter(IConverter<long, TLink>
            ↪   int64ToLongRawNumberConverter) => _int64ToLongRawNumberConverter =
            ↪   int64ToLongRawNumberConverter;
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          public TLink Convert(DateTime source) =>
            ↪   _int64ToLongRawNumberConverter.Convert(source.ToFileTimeUtc());
18      }
19  }
```

## 1.161  ./csharp/Platform.Data.Doublets/Time/LongRawNumberSequenceToDateTimeConverter.cs

```
1   using System;
2   using System.Runtime.CompilerServices;
3   using Platform.Converters;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.Time
8   {
9       public class LongRawNumberSequenceToDateTimeConverter<TLink> : IConverter<TLink, DateTime>
10      {
11          private readonly IConverter<TLink, long> _longRawNumberConverterToInt64;
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          public LongRawNumberSequenceToDateTimeConverter(IConverter<TLink, long>
            ↪   longRawNumberConverterToInt64) => _longRawNumberConverterToInt64 =
            ↪   longRawNumberConverterToInt64;
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          public DateTime Convert(TLink source) =>
            ↪   DateTime.FromFileTimeUtc(_longRawNumberConverterToInt64.Convert(source));
18      }
19  }
```

## 1.162  ./csharp/Platform.Data.Doublets/UInt64LinksExtensions.cs

```
1   using System;
2   using System.Text;
3   using System.Collections.Generic;
4   using System.Runtime.CompilerServices;
5   using Platform.Singletons;
6   using Platform.Data.Doublets.Unicode;
7
8   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10  namespace Platform.Data.Doublets
11  {
12      public static class UInt64LinksExtensions
```

```csharp
{
    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);
```

```
 80                              }
 81                              else
 82                              {
 83                                  var source = new Link<ulong>(links.GetLink(link.Source));
 84                                  if (isElement(source))
 85                                  {
 86                                      appendElement(sb, source);
 87                                  }
 88                                  else
 89                                  {
 90                                      links.AppendStructure(sb, visited, source.Index, isElement,
                                      ↪  appendElement, renderIndex);
 91                                  }
 92                              }
 93                              sb.Append(' ');
 94                              if (link.Target == link.Index)
 95                              {
 96                                  sb.Append(link.Index);
 97                              }
 98                              else
 99                              {
100                                  var target = new Link<ulong>(links.GetLink(link.Target));
101                                  if (isElement(target))
102                                  {
103                                      appendElement(sb, target);
104                                  }
105                                  else
106                                  {
107                                      links.AppendStructure(sb, visited, target.Index, isElement,
                                      ↪  appendElement, renderIndex);
108                                  }
109                              }
110                              sb.Append(')');
111                          }
112                          else
113                          {
114                              if (renderDebug)
115                              {
116                                  sb.Append('*');
117                              }
118                              sb.Append(linkIndex);
119                          }
120                      }
121                      else
122                      {
123                          if (renderDebug)
124                          {
125                              sb.Append('~');
126                          }
127                          sb.Append(linkIndex);
128                      }
129                  }
130              }
131          }
```

## 1.163   ./csharp/Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs

```
 1   using System;
 2   using System.Linq;
 3   using System.Collections.Generic;
 4   using System.IO;
 5   using System.Runtime.CompilerServices;
 6   using System.Threading;
 7   using System.Threading.Tasks;
 8   using Platform.Disposables;
 9   using Platform.Timestamps;
10   using Platform.Unsafe;
11   using Platform.IO;
12   using Platform.Data.Doublets.Decorators;
13   using Platform.Exceptions;
14
15   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
16
17   namespace Platform.Data.Doublets
18   {
19       public class UInt64LinksTransactionsLayer : LinksDisposableDecoratorBase<ulong> //-V3073
20       {
21           /// <remarks>
22           /// Альтернативные варианты хранения трансформации (элемента транзакции):
23           ///
```

```csharp
        /// 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();
```

```csharp
            }

        [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; }
```

```csharp
            [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);
```

```csharp
        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);
```

```csharp
316                    CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
         ↪   deletedLink, default));
317            }
318
319        [MethodImpl(MethodImplOptions.AggressiveInlining)]
320        private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
         ↪   _transitions;
321
322        [MethodImpl(MethodImplOptions.AggressiveInlining)]
323        private void CommitTransition(Transition transition)
324        {
325            if (_currentTransaction != null)
326            {
327                Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
328            }
329            var transitions = GetCurrentTransitions();
330            transitions.Enqueue(transition);
331        }
332
333        [MethodImpl(MethodImplOptions.AggressiveInlining)]
334        private void RevertTransition(Transition transition)
335        {
336            if (transition.After.IsNull()) // Revert Deletion with Creation
337            {
338                _links.Create();
339            }
340            else if (transition.Before.IsNull()) // Revert Creation with Deletion
341            {
342                _links.Delete(transition.After.Index);
343            }
344            else // Revert Update
345            {
346                _links.Update(new[] { transition.After.Index, transition.Before.Source,
         ↪   transition.Before.Target });
347            }
348        }
349
350        [MethodImpl(MethodImplOptions.AggressiveInlining)]
351        private void ResetCurrentTransation()
352        {
353            _currentTransactionId = 0;
354            _currentTransactionTransitions = null;
355            _currentTransaction = null;
356        }
357
358        [MethodImpl(MethodImplOptions.AggressiveInlining)]
359        private void PushTransitions()
360        {
361            if (_log == null || _transitions == null)
362            {
363                return;
364            }
365            for (var i = 0; i < _transitions.Count; i++)
366            {
367                var transition = _transitions.Dequeue();
368
369                _log.Write(transition);
370                _lastCommitedTransition = transition;
371            }
372        }
373
374        [MethodImpl(MethodImplOptions.AggressiveInlining)]
375        private void TransitionsPusher()
376        {
377            while (!Disposable.IsDisposed && _transitionsPusher != null)
378            {
379                Thread.Sleep(DefaultPushDelay);
380                PushTransitions();
381            }
382        }
383
384        [MethodImpl(MethodImplOptions.AggressiveInlining)]
385        public Transaction BeginTransaction() => new Transaction(this);
386
387        [MethodImpl(MethodImplOptions.AggressiveInlining)]
388        private void DisposeTransitions()
389        {
390            try
391            {
```

```
392             var pusher = _transitionsPusher;
393             if (pusher != null)
394             {
395                 _transitionsPusher = null;
396                 pusher.Wait();
397             }
398             if (_transitions != null)
399             {
400                 PushTransitions();
401             }
402             _log.DisposeIfPossible();
403             FileHelpers.WriteFirst(_logAddress, _lastCommitedTransition);
404         }
405         catch (Exception ex)
406         {
407             ex.Ignore();
408         }
409     }
410
411     #region DisposalBase
412
413     [MethodImpl(MethodImplOptions.AggressiveInlining)]
414     protected override void Dispose(bool manual, bool wasDisposed)
415     {
416         if (!wasDisposed)
417         {
418             DisposeTransitions();
419         }
420         base.Dispose(manual, wasDisposed);
421     }
422
423     #endregion
424     }
425 }
```

## 1.164 ./csharp/Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs

```
1  using System.Runtime.CompilerServices;
2  using Platform.Converters;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Unicode
7  {
8      public class CharToUnicodeSymbolConverter<TLink> : LinksOperatorBase<TLink>,
         ↪  IConverter<char, TLink>
9      {
10         private static readonly UncheckedConverter<char, TLink> _charToAddressConverter =
            ↪  UncheckedConverter<char, TLink>.Default;
11
12         private readonly IConverter<TLink> _addressToNumberConverter;
13         private readonly TLink _unicodeSymbolMarker;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public CharToUnicodeSymbolConverter(ILinks<TLink> links, IConverter<TLink>
            ↪  addressToNumberConverter, TLink unicodeSymbolMarker) : base(links)
17         {
18             _addressToNumberConverter = addressToNumberConverter;
19             _unicodeSymbolMarker = unicodeSymbolMarker;
20         }
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public TLink Convert(char source)
24         {
25             var unaryNumber =
                ↪  _addressToNumberConverter.Convert(_charToAddressConverter.Convert(source));
26             return _links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
27         }
28     }
29 }
```

## 1.165 ./csharp/Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Converters;
4  using Platform.Data.Doublets.Sequences.Indexes;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Unicode
9  {
```

```csharp
      public class StringToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
      ↪  IConverter<string, TLink>
      {
          private readonly IConverter<string, IList<TLink>> _stringToUnicodeSymbolListConverter;
          private readonly IConverter<IList<TLink>, TLink> _unicodeSymbolListToSequenceConverter;

          [MethodImpl(MethodImplOptions.AggressiveInlining)]
          public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
          ↪  IList<TLink>> stringToUnicodeSymbolListConverter, IConverter<IList<TLink>, TLink>
          ↪  unicodeSymbolListToSequenceConverter) : base(links)
          {
              _stringToUnicodeSymbolListConverter = stringToUnicodeSymbolListConverter;
              _unicodeSymbolListToSequenceConverter = unicodeSymbolListToSequenceConverter;
          }

          [MethodImpl(MethodImplOptions.AggressiveInlining)]
          public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
          ↪  IList<TLink>> stringToUnicodeSymbolListConverter, ISequenceIndex<TLink> index,
          ↪  IConverter<IList<TLink>, TLink> listToSequenceLinkConverter, TLink
          ↪  unicodeSequenceMarker)
              : this(links, stringToUnicodeSymbolListConverter, new
              ↪  UnicodeSymbolsListToUnicodeSequenceConverter<TLink>(links, index,
              ↪  listToSequenceLinkConverter, unicodeSequenceMarker)) { }

          [MethodImpl(MethodImplOptions.AggressiveInlining)]
          public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
          ↪  charToUnicodeSymbolConverter, ISequenceIndex<TLink> index, IConverter<IList<TLink>,
          ↪  TLink> listToSequenceLinkConverter, TLink unicodeSequenceMarker)
              : this(links, new
              ↪  StringToUnicodeSymbolsListConverter<TLink>(charToUnicodeSymbolConverter), index,
              ↪  listToSequenceLinkConverter, unicodeSequenceMarker) { }

          [MethodImpl(MethodImplOptions.AggressiveInlining)]
          public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
          ↪  charToUnicodeSymbolConverter, IConverter<IList<TLink>, TLink>
          ↪  listToSequenceLinkConverter, TLink unicodeSequenceMarker)
              : this(links, charToUnicodeSymbolConverter, new Unindex<TLink>(),
              ↪  listToSequenceLinkConverter, unicodeSequenceMarker) { }

          [MethodImpl(MethodImplOptions.AggressiveInlining)]
          public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
          ↪  IList<TLink>> stringToUnicodeSymbolListConverter, IConverter<IList<TLink>, TLink>
          ↪  listToSequenceLinkConverter, TLink unicodeSequenceMarker)
              : this(links, stringToUnicodeSymbolListConverter, new Unindex<TLink>(),
              ↪  listToSequenceLinkConverter, unicodeSequenceMarker) { }

          [MethodImpl(MethodImplOptions.AggressiveInlining)]
          public TLink Convert(string source)
          {
              var elements = _stringToUnicodeSymbolListConverter.Convert(source);
              return _unicodeSymbolListToSequenceConverter.Convert(elements);
          }
      }
  }
```

## 1.166 ./csharp/Platform.Data.Doublets/Unicode/StringToUnicodeSymbolsListConverter.cs

```csharp
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.Unicode
{
    public class StringToUnicodeSymbolsListConverter<TLink> : IConverter<string, IList<TLink>>
    {
        private readonly IConverter<char, TLink> _charToUnicodeSymbolConverter;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public StringToUnicodeSymbolsListConverter(IConverter<char, TLink>
        ↪  charToUnicodeSymbolConverter) => _charToUnicodeSymbolConverter =
        ↪  charToUnicodeSymbolConverter;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IList<TLink> Convert(string source)
        {
            var elements = new TLink[source.Length];
            for (var i = 0; i < elements.Length; i++)
            {
```

```
22                elements[i] = _charToUnicodeSymbolConverter.Convert(source[i]);
23            }
24            return elements;
25        }
26    }
27 }
```

**1.167 ./csharp/Platform.Data.Doublets/Unicode/UnicodeMap.cs**

```csharp
1  using System;
2  using System.Collections.Generic;
3  using System.Globalization;
4  using System.Runtime.CompilerServices;
5  using System.Text;
6  using Platform.Data.Sequences;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Unicode
11 {
12     public class UnicodeMap
13     {
14         public static readonly ulong FirstCharLink = 1;
15         public static readonly ulong LastCharLink = FirstCharLink + char.MaxValue;
16         public static readonly ulong MapSize = 1 + char.MaxValue;
17
18         private readonly ILinks<ulong> _links;
19         private bool _initialized;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public UnicodeMap(ILinks<ulong> links) => _links = links;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public static UnicodeMap InitNew(ILinks<ulong> links)
26         {
27             var map = new UnicodeMap(links);
28             map.Init();
29             return map;
30         }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public void Init()
34         {
35             if (_initialized)
36             {
37                 return;
38             }
39             _initialized = true;
40             var firstLink = _links.CreatePoint();
41             if (firstLink != FirstCharLink)
42             {
43                 _links.Delete(firstLink);
44             }
45             else
46             {
47                 for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
48                 {
49                     // From NIL to It (NIL -> Character) transformation meaning, (or infinite
                     ↪    amount of NIL characters before actual Character)
50                     var createdLink = _links.CreatePoint();
51                     _links.Update(createdLink, firstLink, createdLink);
52                     if (createdLink != i)
53                     {
54                         throw new InvalidOperationException("Unable to initialize UTF 16
                         ↪    table.");
55                     }
56                 }
57             }
58         }
59
60         // 0 - null link
61         // 1 - nil character (0 character)
62         // ...
63         // 65536 (0(1) + 65535 = 65536 possible values)
64
65         [MethodImpl(MethodImplOptions.AggressiveInlining)]
66         public static ulong FromCharToLink(char character) => (ulong)character + 1;
67
68         [MethodImpl(MethodImplOptions.AggressiveInlining)]
69         public static char FromLinkToChar(ulong link) => (char)(link - 1);
70
```

```csharp
        [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;
```

```csharp
                    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;
            }
        }
    }
```

## 1.168   ./csharp/Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs

```csharp
using System;
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Converters;
using Platform.Data.Doublets.Sequences.Walkers;
using System.Text;

#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)
        {
```

```csharp
21                    _unicodeSequenceCriterionMatcher = unicodeSequenceCriterionMatcher;
22                    _sequenceWalker = sequenceWalker;
23                    _unicodeSymbolToCharConverter = unicodeSymbolToCharConverter;
24                }
25
26                [MethodImpl(MethodImplOptions.AggressiveInlining)]
27                public string Convert(TLink source)
28                {
29                    if (!_unicodeSequenceCriterionMatcher.IsMatched(source))
30                    {
31                        throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
   ↪   not a unicode sequence.");
32                    }
33                    var sequence = _links.GetSource(source);
34                    var sb = new StringBuilder();
35                    foreach(var character in _sequenceWalker.Walk(sequence))
36                    {
37                        sb.Append(_unicodeSymbolToCharConverter.Convert(character));
38                    }
39                    return sb.ToString();
40                }
41            }
42        }
```

## 1.169  ./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs

```csharp
1   using System;
2   using System.Runtime.CompilerServices;
3   using Platform.Interfaces;
4   using Platform.Converters;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Data.Doublets.Unicode
9   {
10      public class UnicodeSymbolToCharConverter<TLink> : LinksOperatorBase<TLink>,
    ↪   IConverter<TLink, char>
11      {
12          private static readonly UncheckedConverter<TLink, char> _addressToCharConverter =
    ↪   UncheckedConverter<TLink, char>.Default;
13
14          private readonly IConverter<TLink> _numberToAddressConverter;
15          private readonly ICriterionMatcher<TLink> _unicodeSymbolCriterionMatcher;
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          public UnicodeSymbolToCharConverter(ILinks<TLink> links, IConverter<TLink>
    ↪   numberToAddressConverter, ICriterionMatcher<TLink> unicodeSymbolCriterionMatcher) :
    ↪   base(links)
19          {
20              _numberToAddressConverter = numberToAddressConverter;
21              _unicodeSymbolCriterionMatcher = unicodeSymbolCriterionMatcher;
22          }
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          public char Convert(TLink source)
26          {
27              if (!_unicodeSymbolCriterionMatcher.IsMatched(source))
28              {
29                  throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
    ↪   not a unicode symbol.");
30              }
31              return _addressToCharConverter.Convert(_numberToAddressConverter.Convert(_links.GetS
    ↪   ource(source)));
32          }
33      }
34  }
```

## 1.170  ./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolsListToUnicodeSequenceConverter.cs

```csharp
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Converters;
4   using Platform.Data.Doublets.Sequences.Indexes;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Data.Doublets.Unicode
9   {
10      public class UnicodeSymbolsListToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
    ↪   IConverter<IList<TLink>, TLink>
11      {
```

```csharp
        private readonly ISequenceIndex<TLink> _index;
        private readonly IConverter<IList<TLink>, TLink> _listToSequenceLinkConverter;
        private readonly TLink _unicodeSequenceMarker;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnicodeSymbolsListToUnicodeSequenceConverter(ILinks<TLink> links,
            ISequenceIndex<TLink> index, IConverter<IList<TLink>, TLink>
            listToSequenceLinkConverter, TLink unicodeSequenceMarker) : base(links)
        {
            _index = index;
            _listToSequenceLinkConverter = listToSequenceLinkConverter;
            _unicodeSequenceMarker = unicodeSequenceMarker;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnicodeSymbolsListToUnicodeSequenceConverter(ILinks<TLink> links,
            IConverter<IList<TLink>, TLink> listToSequenceLinkConverter, TLink
            unicodeSequenceMarker)
            : this(links, new Unindex<TLink>(), listToSequenceLinkConverter,
                unicodeSequenceMarker) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(IList<TLink> list)
        {
            _index.Add(list);
            var sequence = _listToSequenceLinkConverter.Convert(list);
            return _links.GetOrCreate(sequence, _unicodeSequenceMarker);
        }
    }
}
```

## 1.171 ./csharp/Platform.Data.Doublets.Tests/DefaultSequenceAppenderTests.cs

```csharp
using System.Collections.Generic;
using Platform.Collections.Stacks;
using Platform.Data.Doublets.Memory;
using Platform.Data.Doublets.Memory.United.Generic;
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Sequences.HeightProviders;
using Platform.Data.Numbers.Raw;
using Platform.Interfaces;
using Platform.Memory;
using Platform.Numbers;
using Xunit;
using Xunit.Abstractions;
using TLink = System.UInt64;

namespace Platform.Data.Doublets.Tests
{
    public class DefaultSequenceAppenderTests
    {
        private readonly ITestOutputHelper _output;

        public DefaultSequenceAppenderTests(ITestOutputHelper output)
        {
            _output = output;
        }
        public static ILinks<TLink> CreateLinks() => CreateLinks<TLink>(new IO.TemporaryFile());

        public static ILinks<TLink> CreateLinks<TLink>(string dataDBFilename)
        {
            var linksConstants = new LinksConstants<TLink>(enableExternalReferencesSupport:
                true);
            return new UnitedMemoryLinks<TLink>(new
                FileMappedResizableDirectMemory(dataDBFilename),
                UnitedMemoryLinks<TLink>.DefaultLinksSizeStep, linksConstants,
                IndexTreeType.Default);
        }

        public class ValueCriterionMatcher<TLink> : ICriterionMatcher<TLink>
        {
            public readonly ILinks<TLink> Links;
            public readonly TLink Marker;
            public ValueCriterionMatcher(ILinks<TLink> links, TLink marker)
            {
                Links = links;
                Marker = marker;
            }

            public bool IsMatched(TLink link) =>
                EqualityComparer<TLink>.Default.Equals(Links.GetSource(link), Marker);
```

```
            }

            [Fact]
            public void AppendArrayBug()
            {
                ILinks<TLink> links = CreateLinks();
                TLink zero = default;
                var markerIndex = Arithmetic.Increment(zero);
                var meaningRoot = links.GetOrCreate(markerIndex, markerIndex);
                var sequence = links.Create();
                sequence = links.Update(sequence, meaningRoot, sequence);
                var appendant = links.Create();
                appendant = links.Update(appendant, meaningRoot, appendant);
                ValueCriterionMatcher<TLink> valueCriterionMatcher = new(links, meaningRoot);
                DefaultSequenceRightHeightProvider<ulong> defaultSequenceRightHeightProvider =
                  ↪ new(links, valueCriterionMatcher);
                DefaultSequenceAppender<TLink> defaultSequenceAppender = new(links, new
                  ↪ DefaultStack<ulong>(), defaultSequenceRightHeightProvider);
                var newArray = defaultSequenceAppender.Append(sequence, appendant);
                var output = links.FormatStructure(newArray, link => link.IsFullPoint(), true);
                Assert.Equal("(4:(2:1 2) (3:1 3))", output);
            }
        }
    }
```

## 1.172   ./csharp/Platform.Data.Doublets.Tests/GenericLinksTests.cs

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

```
47    }
```

## 1.173 ./csharp/Platform.Data.Doublets.Tests/ILinksExtensionsTests.cs

```csharp
1   using Xunit;
2
3   namespace Platform.Data.Doublets.Tests
4   {
5       public class ILinksExtensionsTests
6       {
7           [Fact]
8           public void FormatTest()
9           {
10              using (var scope = new TempLinksTestScope())
11              {
12                  var links = scope.Links;
13                  var link = links.Create();
14                  var linkString = links.Format(link);
15                  Assert.Equal("(1: 1 1)", linkString);
16              }
17          }
18      }
19  }
```

## 1.174 ./csharp/Platform.Data.Doublets.Tests/LinksConstantsTests.cs

```csharp
1   using Xunit;
2
3   namespace Platform.Data.Doublets.Tests
4   {
5       public static class LinksConstantsTests
6       {
7           [Fact]
8           public static void ExternalReferencesTest()
9           {
10              LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
                  ↪  (long.MaxValue + 1UL, ulong.MaxValue));
11
12              //var minimum = new Hybrid<ulong>(0, isExternal: true);
13              var minimum = new Hybrid<ulong>(1, isExternal: true);
14              var maximum = new Hybrid<ulong>(long.MaxValue, isExternal: true);
15
16              Assert.True(constants.IsExternalReference(minimum));
17              Assert.True(constants.IsExternalReference(maximum));
18          }
19      }
20  }
```

## 1.175 ./csharp/Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs

```csharp
1   using System;
2   using System.Linq;
3   using Xunit;
4   using Platform.Collections.Stacks;
5   using Platform.Collections.Arrays;
6   using Platform.Memory;
7   using Platform.Data.Numbers.Raw;
8   using Platform.Data.Doublets.Sequences;
9   using Platform.Data.Doublets.Sequences.Frequencies.Cache;
10  using Platform.Data.Doublets.Sequences.Frequencies.Counters;
11  using Platform.Data.Doublets.Sequences.Converters;
12  using Platform.Data.Doublets.PropertyOperators;
13  using Platform.Data.Doublets.Incrementers;
14  using Platform.Data.Doublets.Sequences.Walkers;
15  using Platform.Data.Doublets.Sequences.Indexes;
16  using Platform.Data.Doublets.Unicode;
17  using Platform.Data.Doublets.Numbers.Unary;
18  using Platform.Data.Doublets.Decorators;
19  using Platform.Data.Doublets.Memory.United.Specific;
20  using Platform.Data.Doublets.Memory;
21
22  namespace Platform.Data.Doublets.Tests
23  {
24      public static class OptimalVariantSequenceTests
25      {
26          private static readonly string _sequenceExample = "зеленела зелёная зелень";
27          private static readonly string _loremIpsumExample = @"Lorem ipsum dolor sit amet,
              ↪  consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore
              ↪  magna aliqua.
28  Facilisi nullam vehicula ipsum a arcu cursus vitae congue mauris.
29  Et malesuada fames ac turpis egestas sed.
30  Eget velit aliquet sagittis id consectetur purus.
31  Dignissim cras tincidunt lobortis feugiat vivamus.
```

```
32  Vitae aliquet nec ullamcorper sit.
33  Lectus quam id leo in vitae.
34  Tortor dignissim convallis aenean et tortor at risus viverra adipiscing.
35  Sed risus ultricies tristique nulla aliquet enim tortor at auctor.
36  Integer eget aliquet nibh praesent tristique.
37  Vitae congue eu consequat ac felis donec et odio.
38  Tristique et egestas quis ipsum suspendisse.
39  Suspendisse potenti nullam ac tortor vitae purus faucibus ornare.
40  Nulla facilisi etiam dignissim diam quis enim lobortis scelerisque.
41  Imperdiet proin fermentum leo vel orci.
42  In ante metus dictum at tempor commodo.
43  Nisi lacus sed viverra tellus in.
44  Quam vulputate dignissim suspendisse in.
45  Elit scelerisque mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus.
46  Gravida cum sociis natoque penatibus et magnis dis parturient.
47  Risus quis varius quam quisque id diam.
48  Congue nisi vitae suscipit tellus mauris a diam maecenas.
49  Eget nunc scelerisque viverra mauris in aliquam sem fringilla.
50  Pharetra vel turpis nunc eget lorem dolor sed viverra.
51  Mattis pellentesque id nibh tortor id aliquet.
52  Purus non enim praesent elementum facilisis leo vel.
53  Etiam sit amet nisl purus in mollis nunc sed.
54  Tortor at auctor urna nunc id cursus metus aliquam.
55  Volutpat odio facilisis mauris sit amet.
56  Turpis egestas pretium aenean pharetra magna ac placerat.
57  Fermentum dui faucibus in ornare quam viverra orci sagittis eu.
58  Porttitor leo a diam sollicitudin tempor id eu.
59  Volutpat sed cras ornare arcu dui.
60  Ut aliquam purus sit amet luctus venenatis lectus magna.
61  Aliquet risus feugiat in ante metus dictum at.
62  Mattis nunc sed blandit libero.
63  Elit pellentesque habitant morbi tristique senectus et netus.
64  Nibh sit amet commodo nulla facilisi nullam vehicula ipsum a.
65  Enim sit amet venenatis urna cursus eget nunc scelerisque viverra.
66  Amet venenatis urna cursus eget nunc scelerisque viverra mauris in.
67  Diam donec adipiscing tristique risus nec feugiat.
68  Pulvinar mattis nunc sed blandit libero volutpat.
69  Cras fermentum odio eu feugiat pretium nibh ipsum.
70  In nulla posuere sollicitudin aliquam ultrices sagittis orci a.
71  Mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus et.
72  A iaculis at erat pellentesque.
73  Morbi blandit cursus risus at ultrices mi tempus imperdiet nulla.
74  Eget lorem dolor sed viverra ipsum nunc.
75  Leo a diam sollicitudin tempor id eu.
76  Interdum consectetur libero id faucibus nisl tincidunt eget nullam non.";
77
78          [Fact]
79          public static void LinksBasedFrequencyStoredOptimalVariantSequenceTest()
80          {
81              using (var scope = new TempLinksTestScope(useSequences: false))
82              {
83                  var links = scope.Links;
84                  var constants = links.Constants;
85
86                  links.UseUnicode();
87
88                  var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);
89
90                  var meaningRoot = links.CreatePoint();
91                  var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
92                  var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
93                  var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
                    ↪  constants.Itself);
94
95                  var unaryNumberToAddressConverter = new
                    ↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
96                  var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
97                  var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                    ↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
98                  var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
                    ↪  frequencyPropertyMarker, frequencyMarker);
99                  var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
                    ↪  frequencyPropertyOperator, frequencyIncrementer);
100                 var linkToItsFrequencyNumberConverter = new
                    ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                    ↪  unaryNumberToAddressConverter);
101                 var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                    ↪  linkToItsFrequencyNumberConverter);
102                 var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                    ↪  sequenceToItsLocalElementLevelsConverter);
```

```csharp
                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 FrequenciesCacheBasedLinkToItsFreque↓
                ↪  ncyNumberConverter<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, IndexTreeType.Default))
            {
                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));
```

```
164              var unicodeSequenceMarker = links.GetOrCreate(root,
          ↪    addressToNumberConverter.Convert(2));
165
166              var totalSequenceSymbolFrequencyCounter = new
          ↪    TotalSequenceSymbolFrequencyCounter<ulong>(links);
167              var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
          ↪    totalSequenceSymbolFrequencyCounter);
168              var index = new
          ↪    CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
169              var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque ⌋
          ↪    ncyNumberConverter<ulong>(linkFrequenciesCache);
170              var sequenceToItsLocalElementLevelsConverter = new
          ↪    SequenceToItsLocalElementLevelsConverter<ulong>(links,
          ↪    linkToItsFrequencyNumberConverter);
171              var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
          ↪    sequenceToItsLocalElementLevelsConverter);
172
173              var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>(),
          ↪    (link) => constants.IsExternalReference(link) || links.IsPartialPoint(link));
174
175              var unicodeSequencesOptions = new SequencesOptions<ulong>()
176              {
177                  UseSequenceMarker = true,
178                  SequenceMarkerLink = unicodeSequenceMarker,
179                  UseIndex = true,
180                  Index = index,
181                  LinksToSequenceConverter = optimalVariantConverter,
182                  Walker = walker,
183                  UseGarbageCollection = true
184              };
185
186              var unicodeSequences = new Sequences.Sequences(new
          ↪    SynchronizedLinks<ulong>(links), unicodeSequencesOptions);
187
188              // Create some sequences
189              var strings = _loremIpsumExample.Split(new[] { '\n', '\r' },
          ↪    StringSplitOptions.RemoveEmptyEntries);
190              var arrays = strings.Select(x => x.Select(y =>
          ↪    addressToNumberConverter.Convert(y)).ToArray()).ToArray();
191              for (int i = 0; i < arrays.Length; i++)
192              {
193                  unicodeSequences.Create(arrays[i].ShiftRight());
194              }
195
196              var linksCountAfterCreation = links.Count();
197
198              // get list of sequences links
199              // for each sequence link
200              //   create new sequence version
201              //   if new sequence is not the same as sequence link
202              //     delete sequence link
203              //     collect garbadge
204              unicodeSequences.CompactAll();
205
206              var linksCountAfterCompactification = links.Count();
207
208              Assert.True(linksCountAfterCompactification < linksCountAfterCreation);
209          }
210      }
211  }
212 }
```

## 1.176   ./csharp/Platform.Data.Doublets.Tests/ReadSequenceTests.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Diagnostics;
4   using System.Linq;
5   using Xunit;
6   using Platform.Data.Sequences;
7   using Platform.Data.Doublets.Sequences.Converters;
8   using Platform.Data.Doublets.Sequences.Walkers;
9   using Platform.Data.Doublets.Sequences;
10
11  namespace Platform.Data.Doublets.Tests
12  {
13      public static class ReadSequenceTests
14      {
15          [Fact]
16          public static void ReadSequenceTest()
```

```
17            {
18                const long sequenceLength = 2000;
19
20                using (var scope = new TempLinksTestScope(useSequences: false))
21                {
22                    var links = scope.Links;
23                    var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong> {
                     ↪   Walker = new LeveledSequenceWalker<ulong>(links) });
24
25                    var sequence = new ulong[sequenceLength];
26                    for (var i = 0; i < sequenceLength; i++)
27                    {
28                        sequence[i] = links.Create();
29                    }
30
31                    var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
32
33                    var sw1 = Stopwatch.StartNew();
34                    var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
35
36                    var sw2 = Stopwatch.StartNew();
37                    var readSequence1 = sequences.ToList(balancedVariant); sw2.Stop();
38
39                    var sw3 = Stopwatch.StartNew();
40                    var readSequence2 = new List<ulong>();
41                    SequenceWalker.WalkRight(balancedVariant,
42                                             links.GetSource,
43                                             links.GetTarget,
44                                             links.IsPartialPoint,
45                                             readSequence2.Add);
46                    sw3.Stop();
47
48                    Assert.True(sequence.SequenceEqual(readSequence1));
49
50                    Assert.True(sequence.SequenceEqual(readSequence2));
51
52                    // Assert.True(sw2.Elapsed < sw3.Elapsed);
53
54                    Console.WriteLine($"Stack-based walker: {sw3.Elapsed}, Level-based reader:
                     ↪   {sw2.Elapsed}");
55
56                    for (var i = 0; i < sequenceLength; i++)
57                    {
58                        links.Delete(sequence[i]);
59                    }
60                }
61            }
62        }
63    }
```

## 1.177 ./csharp/Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs

```
1    using System.IO;
2    using Xunit;
3    using Platform.Singletons;
4    using Platform.Memory;
5    using Platform.Data.Doublets.Memory.United.Specific;
6
7    namespace Platform.Data.Doublets.Tests
8    {
9        public static class ResizableDirectMemoryLinksTests
10       {
11           private static readonly LinksConstants<ulong> _constants =
              ↪   Default<LinksConstants<ulong>>.Instance;
12
13           [Fact]
14           public static void BasicFileMappedMemoryTest()
15           {
16               var tempFilename = Path.GetTempFileName();
17               using (var memoryAdapter = new UInt64UnitedMemoryLinks(tempFilename))
18               {
19                   memoryAdapter.TestBasicMemoryOperations();
20               }
21               File.Delete(tempFilename);
22           }
23
24           [Fact]
25           public static void BasicHeapMemoryTest()
26           {
27               using (var memory = new
              ↪   HeapResizableDirectMemory(UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
```

```csharp
                        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);
        }
    }
}
```

## 1.178 ./csharp/Platform.Data.Doublets.Tests/ScopeTests.cs

```csharp
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(Skip = "Would be fixed later.")]
```

```csharp
            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);
                }
            }
        }
    }
```

## 1.179 ./csharp/Platform.Data.Doublets.Tests/SequencesTests.cs

```csharp
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]);
```

```csharp
                }

                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();
```

```csharp
                        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++)
                    {
```

```csharp
                        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();
```

```csharp
288             var searchResults2 =
                ↪    sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
289
290             Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
291
292             Assert.True(searchResults2.Count == 1 && balancedVariant ==
                ↪    searchResults2.First());
293
294             for (var i = 0; i < sequenceLength; i++)
295             {
296                 links.Delete(sequence[i]);
297             }
298         }
299     }
300
301     [Fact(Skip = "Correct implementation is pending")]
302     public static void PatternMatchTest()
303     {
304         var zeroOrMany = Sequences.Sequences.ZeroOrMany;
305
306         using (var scope = new TempLinksTestScope(useSequences: true))
307         {
308             var links = scope.Links;
309             var sequences = scope.Sequences;
310
311             var e1 = links.Create();
312             var e2 = links.Create();
313
314             var sequence = new[]
315             {
316                 e1, e2, e1, e2 // mama / papa
317             };
318
319             var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
320
321             var balancedVariant = balancedVariantConverter.Convert(sequence);
322
323             // 1: [1]
324             // 2: [2]
325             // 3: [1,2]
326             // 4: [1,2,1,2]
327
328             var doublet = links.GetSource(balancedVariant);
329
330             var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);
331
332             Assert.True(matchedSequences1.Count == 0);
333
334             var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);
335
336             Assert.True(matchedSequences2.Count == 0);
337
338             var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);
339
340             Assert.True(matchedSequences3.Count == 0);
341
342             var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
343
344             Assert.Contains(doublet, matchedSequences4);
345             Assert.Contains(balancedVariant, matchedSequences4);
346
347             for (var i = 0; i < sequence.Length; i++)
348             {
349                 links.Delete(sequence[i]);
350             }
351         }
352     }
353
354     [Fact]
355     public static void IndexTest()
356     {
357         using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
                ↪    true }, useSequences: true))
358         {
359             var links = scope.Links;
360             var sequences = scope.Sequences;
361             var index = sequences.Options.Index;
362
363             var e1 = links.Create();
364             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/Links
↪ Platform/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)

```
407  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?
408
409  [![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)
410
411  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?
412
413  [![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)
414
415  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?
416
417  [![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)
418
419  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?
420
421  [![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.c
   ↪  om/Konard/LinksPlatform/master/doc/Intro/10.png)
422
423  На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом
   ↪  рекурсии или фрактала?
424
425  [![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.githubuserc
   ↪  ontent.com/Konard/LinksPlatform/master/doc/Intro/11.png)
426
427  Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы? Буквы?
   ↪  Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?
428
429  [![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)
430
431  ...
432
433  [![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-anima
   ↪  tion-500.gif
   ↪  ""анимация"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro
   ↪  -animation-500.gif)";
434
435          private static readonly string _exampleLoremIpsumText =
436              @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
                  ↪  incididunt ut labore et dolore magna aliqua.
437  Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
   ↪  consequat.";
438
439          [Fact]
440          public static void CompressionTest()
441          {
442              using (var scope = new TempLinksTestScope(useSequences: true))
443              {
```

```
444          var links = scope.Links;
445          var sequences = scope.Sequences;
446
447          var e1 = links.Create();
448          var e2 = links.Create();
449
450          var sequence = new[]
451          {
452              e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
453          };
454
455          var balancedVariantConverter = new BalancedVariantConverter<ulong>(links.Unsync);
456          var totalSequenceSymbolFrequencyCounter = new
             ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
457          var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync,
             ↪ totalSequenceSymbolFrequencyCounter);
458          var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
             ↪ balancedVariantConverter, doubletFrequenciesCache);
459
460          var compressedVariant = compressingConverter.Convert(sequence);
461
462          // 1: [1]         (1->1) point
463          // 2: [2]         (2->2) point
464          // 3: [1,2]       (1->2) doublet
465          // 4: [1,2,1,2]   (3->3) doublet
466
467          Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
468          Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
469          Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
470          Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);
471
472          var source = _constants.SourcePart;
473          var target = _constants.TargetPart;
474
475          Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
476          Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
477          Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
478          Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);
479
480          // 4 - length of sequence
481          Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0)
             ↪ == sequence[0]);
482          Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
             ↪ == sequence[1]);
483          Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
             ↪ == sequence[2]);
484          Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
             ↪ == sequence[3]);
485      }
486  }
487
488  [Fact]
489  public static void CompressionEfficiencyTest()
490  {
491      var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
         ↪ StringSplitOptions.RemoveEmptyEntries);
492      var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
493      var totalCharacters = arrays.Select(x => x.Length).Sum();
494
495      using (var scope1 = new TempLinksTestScope(useSequences: true))
496      using (var scope2 = new TempLinksTestScope(useSequences: true))
497      using (var scope3 = new TempLinksTestScope(useSequences: true))
498      {
499          scope1.Links.Unsync.UseUnicode();
500          scope2.Links.Unsync.UseUnicode();
501          scope3.Links.Unsync.UseUnicode();
502
503          var balancedVariantConverter1 = new
             ↪ BalancedVariantConverter<ulong>(scope1.Links.Unsync);
504          var totalSequenceSymbolFrequencyCounter = new
             ↪ TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
505          var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
             ↪ totalSequenceSymbolFrequencyCounter);
506          var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
             ↪ balancedVariantConverter1, linkFrequenciesCache1,
             ↪ doInitialFrequenciesIncrement: false);
507
508          //var compressor2 = scope2.Sequences;
```

```
509        var compressor3 = scope3.Sequences;

510
511        var constants = Default<LinksConstants<ulong>>.Instance;

512
513        var sequences = compressor3;
514        //var meaningRoot = links.CreatePoint();
515        //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
516        //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
517        //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
    ↪    constants.Itself);
518
519        //var unaryNumberToAddressConverter = new
    ↪    UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
520        //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
    ↪    unaryOne);
521        //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
    ↪    frequencyMarker, unaryOne, unaryNumberIncrementer);
522        //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
    ↪    frequencyPropertyMarker, frequencyMarker);
523        //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
    ↪    frequencyPropertyOperator, frequencyIncrementer);
524        //var linkToItsFrequencyNumberConverter = new
    ↪    LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
    ↪    unaryNumberToAddressConverter);
525
526        var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
    ↪    totalSequenceSymbolFrequencyCounter);
527
528        var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque⌋
    ↪    ncyNumberConverter<ulong>(linkFrequenciesCache3);
529
530        var sequenceToItsLocalElementLevelsConverter = new
    ↪    SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
    ↪    linkToItsFrequencyNumberConverter);
531        var optimalVariantConverter = new
    ↪    OptimalVariantConverter<ulong>(scope3.Links.Unsync,
    ↪    sequenceToItsLocalElementLevelsConverter);
532
533        var compressed1 = new ulong[arrays.Length];
534        var compressed2 = new ulong[arrays.Length];
535        var compressed3 = new ulong[arrays.Length];

536
537        var START = 0;
538        var END = arrays.Length;

539
540        //for (int i = START; i < END; i++)
541        //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);

542
543        var initialCount1 = scope2.Links.Unsync.Count();

544
545        var sw1 = Stopwatch.StartNew();

546
547        for (int i = START; i < END; i++)
548        {
549            linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
550            compressed1[i] = compressor1.Convert(arrays[i]);
551        }

552
553        var elapsed1 = sw1.Elapsed;

554
555        var balancedVariantConverter2 = new
    ↪    BalancedVariantConverter<ulong>(scope2.Links.Unsync);

556
557        var initialCount2 = scope2.Links.Unsync.Count();

558
559        var sw2 = Stopwatch.StartNew();

560
561        for (int i = START; i < END; i++)
562        {
563            compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
564        }

565
566        var elapsed2 = sw2.Elapsed;

567
568        for (int i = START; i < END; i++)
569        {
570            linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
571        }

572
```

```
573                    var initialCount3 = scope3.Links.Unsync.Count();
574
575                    var sw3 = Stopwatch.StartNew();
576
577                    for (int i = START; i < END; i++)
578                    {
579                        //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
580                        compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
581                    }
582
583                    var elapsed3 = sw3.Elapsed;
584
585                    Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
       ↪  Optimal variant: {elapsed3}");
586
587                    // Assert.True(elapsed1 > elapsed2);
588
589                    // Checks
590                    for (int i = START; i < END; i++)
591                    {
592                        var sequence1 = compressed1[i];
593                        var sequence2 = compressed2[i];
594                        var sequence3 = compressed3[i];
595
596                        var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
       ↪  scope1.Links.Unsync);
597
598                        var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
       ↪  scope2.Links.Unsync);
599
600                        var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
       ↪  scope3.Links.Unsync);
601
602                        var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
       ↪  link.IsPartialPoint());
603                        var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
       ↪  link.IsPartialPoint());
604                        var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
       ↪  link.IsPartialPoint());
605
606                        //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
       ↪  arrays[i].Length > 3)
607                        //    Assert.False(structure1 == structure2);
608                        //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
       ↪  arrays[i].Length > 3)
609                        //    Assert.False(structure3 == structure2);
610
611                        Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
612                        Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
613                    }
614
615                    Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
       ↪  totalCharacters);
616                    Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
       ↪  totalCharacters);
617                    Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
       ↪  totalCharacters);
618
619                    Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
       ↪  totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) /
       ↪  totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) /
       ↪  totalCharacters}");
620
621                    Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
       ↪  scope2.Links.Unsync.Count() - initialCount2);
622                    Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
       ↪  scope2.Links.Unsync.Count() - initialCount2);
623
624                    var duplicateProvider1 = new
       ↪  DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
625                    var duplicateProvider2 = new
       ↪  DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
626                    var duplicateProvider3 = new
       ↪  DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
627
628                    var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
629                    var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
```

```csharp
                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)
```

```csharp
                    {
                        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;
```

```csharp
            //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);
                    }
                }
            }
```

```
853            Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
854            Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
855
856            Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
           ↪  totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
           ↪  totalCharacters}");
857
858            // Can be worse than balanced variant
859            //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
860
861            //compressor1.ValidateFrequencies();
862        }
863    }
864
865    [Fact]
866    public static void AllTreeBreakDownAtSequencesCreationBugTest()
867    {
868        // Made out of AllPossibleConnectionsTest test.
869
870        //const long sequenceLength = 5; //100% bug
871        const long sequenceLength = 4; //100% bug
872        //const long sequenceLength = 3; //100% _no_bug (ok)
873
874        using (var scope = new TempLinksTestScope(useSequences: true))
875        {
876            var links = scope.Links;
877            var sequences = scope.Sequences;
878
879            var sequence = new ulong[sequenceLength];
880            for (var i = 0; i < sequenceLength; i++)
881            {
882                sequence[i] = links.Create();
883            }
884
885            var createResults = sequences.CreateAllVariants2(sequence);
886
887            Global.Trash = createResults;
888
889            for (var i = 0; i < sequenceLength; i++)
890            {
891                links.Delete(sequence[i]);
892            }
893        }
894    }
895
896    [Fact]
897    public static void AllPossibleConnectionsTest()
898    {
899        const long sequenceLength = 5;
900
901        using (var scope = new TempLinksTestScope(useSequences: true))
902        {
903            var links = scope.Links;
904            var sequences = scope.Sequences;
905
906            var sequence = new ulong[sequenceLength];
907            for (var i = 0; i < sequenceLength; i++)
908            {
909                sequence[i] = links.Create();
910            }
911
912            var createResults = sequences.CreateAllVariants2(sequence);
913            var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
914
915            for (var i = 0; i < 1; i++)
916            {
917                var sw1 = Stopwatch.StartNew();
918                var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
919
920                var sw2 = Stopwatch.StartNew();
921                var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();
922
923                var sw3 = Stopwatch.StartNew();
924                var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();
925
926                var sw4 = Stopwatch.StartNew();
927                var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();
928
929                Global.Trash = searchResults3;
930                Global.Trash = searchResults4; //-V3008
```

```
931
932                        var intersection1 = createResults.Intersect(searchResults1).ToList();
933                        Assert.True(intersection1.Count == createResults.Length);
934
935                        var intersection2 = reverseResults.Intersect(searchResults1).ToList();
936                        Assert.True(intersection2.Count == reverseResults.Length);
937
938                        var intersection0 = searchResults1.Intersect(searchResults2).ToList();
939                        Assert.True(intersection0.Count == searchResults2.Count);
940
941                        var intersection3 = searchResults2.Intersect(searchResults3).ToList();
942                        Assert.True(intersection3.Count == searchResults3.Count);
943
944                        var intersection4 = searchResults3.Intersect(searchResults4).ToList();
945                        Assert.True(intersection4.Count == searchResults4.Count);
946                    }
947
948                    for (var i = 0; i < sequenceLength; i++)
949                    {
950                        links.Delete(sequence[i]);
951                    }
952                }
953            }
954
955            [Fact(Skip = "Correct implementation is pending")]
956            public static void CalculateAllUsagesTest()
957            {
958                const long sequenceLength = 3;
959
960                using (var scope = new TempLinksTestScope(useSequences: true))
961                {
962                    var links = scope.Links;
963                    var sequences = scope.Sequences;
964
965                    var sequence = new ulong[sequenceLength];
966                    for (var i = 0; i < sequenceLength; i++)
967                    {
968                        sequence[i] = links.Create();
969                    }
970
971                    var createResults = sequences.CreateAllVariants2(sequence);
972
973                    //var reverseResults =
974         ↪    sequences.CreateAllVariants2(sequence.Reverse().ToArray());
975                    for (var i = 0; i < 1; i++)
976                    {
977                        var linksTotalUsages1 = new ulong[links.Count() + 1];
978
979                        sequences.CalculateAllUsages(linksTotalUsages1);
980
981                        var linksTotalUsages2 = new ulong[links.Count() + 1];
982
983                        sequences.CalculateAllUsages2(linksTotalUsages2);
984
985                        var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
986                        Assert.True(intersection1.Count == linksTotalUsages2.Length);
987                    }
988
989                    for (var i = 0; i < sequenceLength; i++)
990                    {
991                        links.Delete(sequence[i]);
992                    }
993                }
994            }
995        }
996    }
```

## 1.180 ./csharp/Platform.Data.Doublets.Tests/SplitMemoryGenericLinksTests.cs

```
1  using System;
2  using Xunit;
3  using Platform.Memory;
4  using Platform.Data.Doublets.Memory.Split.Generic;
5  using Platform.Data.Doublets.Memory;
6
7  namespace Platform.Data.Doublets.Tests
8  {
9      public unsafe static class SplitMemoryGenericLinksTests
10     {
11         [Fact]
```

```csharp
        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);
            }
        }
    }
}
```

## 1.181 ./csharp/Platform.Data.Doublets.Tests/SplitMemoryUInt32LinksTests.cs

```csharp
using System;
using Xunit;
using Platform.Memory;
using Platform.Data.Doublets.Memory.Split.Specific;
using TLink = System.UInt32;

namespace Platform.Data.Doublets.Tests
{
    public unsafe static class SplitMemoryUInt32LinksTests
    {
        [Fact]
        public static void CRUDTest()
        {
            Using(links => links.TestCRUDOperations());
        }

        [Fact]
        public static void RawNumbersCRUDTest()
        {
            UsingWithExternalReferences(links => links.TestRawNumbersCRUDOperations());
        }
```

```
23          [Fact]
24          public static void MultipleRandomCreationsAndDeletionsTest()
25          {
26              Using(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultip↵
                 ↪  leRandomCreationsAndDeletions(500));
27          }
28
29          private static void Using(Action<ILinks<TLink>> action)
30          {
31              using (var dataMemory = new HeapResizableDirectMemory())
32              using (var indexMemory = new HeapResizableDirectMemory())
33              using (var memory = new UInt32SplitMemoryLinks(dataMemory, indexMemory))
34              {
35                  action(memory);
36              }
37          }
38
39          private static void UsingWithExternalReferences(Action<ILinks<TLink>> action)
40          {
41              var contants = new LinksConstants<TLink>(enableExternalReferencesSupport: true);
42              using (var dataMemory = new HeapResizableDirectMemory())
43              using (var indexMemory = new HeapResizableDirectMemory())
44              using (var memory = new UInt32SplitMemoryLinks(dataMemory, indexMemory,
                 ↪  UInt32SplitMemoryLinks.DefaultLinksSizeStep, contants))
45              {
46                  action(memory);
47              }
48          }
49      }
50  }
```

## 1.182 ./csharp/Platform.Data.Doublets.Tests/SplitMemoryUInt64LinksTests.cs

```
1   using System;
2   using Xunit;
3   using Platform.Memory;
4   using Platform.Data.Doublets.Memory.Split.Specific;
5   using TLink = System.UInt64;
6
7   namespace Platform.Data.Doublets.Tests
8   {
9       public unsafe static class SplitMemoryUInt64LinksTests
10      {
11          [Fact]
12          public static void CRUDTest()
13          {
14              Using(links => links.TestCRUDOperations());
15          }
16
17          [Fact]
18          public static void RawNumbersCRUDTest()
19          {
20              UsingWithExternalReferences(links => links.TestRawNumbersCRUDOperations());
21          }
22
23          [Fact]
24          public static void MultipleRandomCreationsAndDeletionsTest()
25          {
26              Using(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultip↵
                 ↪  leRandomCreationsAndDeletions(500));
27          }
28
29          private static void Using(Action<ILinks<TLink>> action)
30          {
31              using (var dataMemory = new HeapResizableDirectMemory())
32              using (var indexMemory = new HeapResizableDirectMemory())
33              using (var memory = new UInt64SplitMemoryLinks(dataMemory, indexMemory))
34              {
35                  action(memory);
36              }
37          }
38
39          private static void UsingWithExternalReferences(Action<ILinks<TLink>> action)
40          {
41              var contants = new LinksConstants<TLink>(enableExternalReferencesSupport: true);
42              using (var dataMemory = new HeapResizableDirectMemory())
43              using (var indexMemory = new HeapResizableDirectMemory())
44              using (var memory = new UInt64SplitMemoryLinks(dataMemory, indexMemory,
                 ↪  UInt64SplitMemoryLinks.DefaultLinksSizeStep, contants))
45              {
```

```
46              action(memory);
47          }
48      }
49  }
50 }
```

## 1.183 ./csharp/Platform.Data.Doublets.Tests/TempLinksTestScope.cs

```csharp
1  using System.IO;
2  using Platform.Disposables;
3  using Platform.Data.Doublets.Sequences;
4  using Platform.Data.Doublets.Decorators;
5  using Platform.Data.Doublets.Memory.United.Specific;
6  using Platform.Data.Doublets.Memory.Split.Specific;
7  using Platform.Memory;
8
9  namespace Platform.Data.Doublets.Tests
10 {
11     public class TempLinksTestScope : DisposableBase
12     {
13         public ILinks<ulong> MemoryAdapter { get; }
14         public SynchronizedLinks<ulong> Links { get; }
15         public Sequences.Sequences Sequences { get; }
16         public string TempFilename { get; }
17         public string TempTransactionLogFilename { get; }
18         private readonly bool _deleteFiles;
19
20         public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool
            useLog = false) : this(new SequencesOptions<ulong>(), deleteFiles, useSequences,
            useLog) { }
21
22         public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles =
            true, bool useSequences = false, bool useLog = false)
23         {
24             _deleteFiles = deleteFiles;
25             TempFilename = Path.GetTempFileName();
26             TempTransactionLogFilename = Path.GetTempFileName();
27             //var coreMemoryAdapter = new UInt64UnitedMemoryLinks(TempFilename);
28             var coreMemoryAdapter = new UInt64SplitMemoryLinks(new
                 FileMappedResizableDirectMemory(TempFilename), new
                 FileMappedResizableDirectMemory(Path.ChangeExtension(TempFilename, "indexes")),
                 UInt64SplitMemoryLinks.DefaultLinksSizeStep, new LinksConstants<ulong>(),
                 Memory.IndexTreeType.Default, useLinkedList: true);
29             MemoryAdapter = useLog ? (ILinks<ulong>)new
                 UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
                 coreMemoryAdapter;
30             Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
31             if (useSequences)
32             {
33                 Sequences = new Sequences.Sequences(Links, sequencesOptions);
34             }
35         }
36
37         protected override void Dispose(bool manual, bool wasDisposed)
38         {
39             if (!wasDisposed)
40             {
41                 Links.Unsync.DisposeIfPossible();
42                 if (_deleteFiles)
43                 {
44                     DeleteFiles();
45                 }
46             }
47         }
48
49         public void DeleteFiles()
50         {
51             File.Delete(TempFilename);
52             File.Delete(TempTransactionLogFilename);
53         }
54     }
55 }
```

## 1.184 ./csharp/Platform.Data.Doublets.Tests/TestExtensions.cs

```csharp
1  using System.Collections.Generic;
2  using Xunit;
3  using Platform.Ranges;
4  using Platform.Numbers;
5  using Platform.Random;
6  using Platform.Setters;
```

```csharp
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);
```

```
 87           var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
 88           var h108E = new Hybrid<T>(-108L);
 89
 90       Assert.Equal(106L, h106E.AbsoluteValue);
 91       Assert.Equal(107L, h107E.AbsoluteValue);
 92       Assert.Equal(108L, h108E.AbsoluteValue);
 93
 94       // Create Link (External -> External)
 95       var linkAddress1 = links.Create();
 96
 97       links.Update(linkAddress1, h106E, h108E);
 98
 99       var link1 = new Link<T>(links.GetLink(linkAddress1));
100
101       Assert.True(equalityComparer.Equals(link1.Source, h106E));
102       Assert.True(equalityComparer.Equals(link1.Target, h108E));
103
104       // Create Link (Internal -> External)
105       var linkAddress2 = links.Create();
106
107       links.Update(linkAddress2, linkAddress1, h108E);
108
109       var link2 = new Link<T>(links.GetLink(linkAddress2));
110
111       Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
112       Assert.True(equalityComparer.Equals(link2.Target, h108E));
113
114       // Create Link (Internal -> Internal)
115       var linkAddress3 = links.Create();
116
117       links.Update(linkAddress3, linkAddress1, linkAddress2);
118
119       var link3 = new Link<T>(links.GetLink(linkAddress3));
120
121       Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
122       Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));
123
124       // Search for created link
125       var setter1 = new Setter<T>(constants.Null);
126       links.Each(h106E, h108E, setter1.SetAndReturnFalse);
127
128       Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));
129
130       // Search for nonexistent link
131       var setter2 = new Setter<T>(constants.Null);
132       links.Each(h106E, h107E, setter2.SetAndReturnFalse);
133
134       Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));
135
136       // Update link to reference null (prepare for delete)
137       var updated = links.Update(linkAddress3, constants.Null, constants.Null);
138
139       Assert.True(equalityComparer.Equals(updated, linkAddress3));
140
141       link3 = new Link<T>(links.GetLink(linkAddress3));
142
143       Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
144       Assert.True(equalityComparer.Equals(link3.Target, constants.Null));
145
146       // Delete link
147       links.Delete(linkAddress3);
148
149       Assert.True(equalityComparer.Equals(links.Count(), two));
150
151       var setter3 = new Setter<T>(constants.Null);
152       links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);
153
154       Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
155   }
156
157   public static void TestMultipleRandomCreationsAndDeletions<TLink>(this ILinks<TLink>
       ↪  links, int maximumOperationsPerCycle)
158   {
159       var comparer = Comparer<TLink>.Default;
160       var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
161       var uInt64ToAddressConverter = CheckedConverter<ulong, TLink>.Default;
162       for (var N = 1; N < maximumOperationsPerCycle; N++)
163       {
164           var random = new System.Random(N);
165           var created = 0UL;
```

```
166                         var deleted = 0UL;
167                         for (var i = 0; i < N; i++)
168                         {
169                             var linksCount = addressToUInt64Converter.Convert(links.Count());
170                             var createPoint = random.NextBoolean();
171                             if (linksCount >= 2 && createPoint)
172                             {
173                                 var linksAddressRange = new Range<ulong>(1, linksCount);
174                                 TLink source = uInt64ToAddressConverter.Convert(random.NextUInt64(linksA
                                   ↪   ddressRange));
175                                 TLink target = uInt64ToAddressConverter.Convert(random.NextUInt64(linksA
                                   ↪   ddressRange));
                                   ↪   //-V3086
176                                 var resultLink = links.GetOrCreate(source, target);
177                                 if (comparer.Compare(resultLink,
                                   ↪   uInt64ToAddressConverter.Convert(linksCount)) > 0)
178                                 {
179                                     created++;
180                                 }
181                             }
182                             else
183                             {
184                                 links.Create();
185                                 created++;
186                             }
187                         }
188                         Assert.True(created == addressToUInt64Converter.Convert(links.Count()));
189                         for (var i = 0; i < N; i++)
190                         {
191                             TLink link = uInt64ToAddressConverter.Convert((ulong)i + 1UL);
192                             if (links.Exists(link))
193                             {
194                                 links.Delete(link);
195                                 deleted++;
196                             }
197                         }
198                         Assert.True(addressToUInt64Converter.Convert(links.Count()) == 0L);
199                     }
200                 }
201             }
202     }
```

## 1.185 ./csharp/Platform.Data.Doublets.Tests/UInt64LinksTests.cs

```
1    using System;
2    using System.Collections.Generic;
3    using System.Diagnostics;
4    using System.IO;
5    using System.Text;
6    using System.Threading;
7    using System.Threading.Tasks;
8    using Xunit;
9    using Platform.Disposables;
10   using Platform.Ranges;
11   using Platform.Random;
12   using Platform.Timestamps;
13   using Platform.Reflection;
14   using Platform.Singletons;
15   using Platform.Scopes;
16   using Platform.Counters;
17   using Platform.Diagnostics;
18   using Platform.IO;
19   using Platform.Memory;
20   using Platform.Data.Doublets.Decorators;
21   using Platform.Data.Doublets.Memory.United.Specific;
22
23   namespace Platform.Data.Doublets.Tests
24   {
25       public static class UInt64LinksTests
26       {
27           private static readonly LinksConstants<ulong> _constants =
             ↪   Default<LinksConstants<ulong>>.Instance;
28
29           private const long Iterations = 10 * 1024;
30
31           #region Concept
32
33           [Fact]
34           public static void MultipleCreateAndDeleteTest()
35           {
```

```
36      using (var scope = new Scope<Types<HeapResizableDirectMemory,
        ↪   UInt64UnitedMemoryLinks>>())
37      {
38          new UInt64Links(scope.Use<ILinks<ulong>>()).TestMultipleRandomCreationsAndDeleti⌐
            ↪   ons(100);
39      }
40  }

41
42  [Fact]
43  public static void CascadeUpdateTest()
44  {
45      var itself = _constants.Itself;
46      using (var scope = new TempLinksTestScope(useLog: true))
47      {
48          var links = scope.Links;
49
50          var l1 = links.Create();
51          var l2 = links.Create();
52
53          l2 = links.Update(l2, l2, l1, l2);
54
55          links.CreateAndUpdate(l2, itself);
56          links.CreateAndUpdate(l2, itself);
57
58          l2 = links.Update(l2, l1);
59
60          links.Delete(l2);
61
62          Global.Trash = links.Count();
63
64          links.Unsync.DisposeIfPossible(); // Close links to access log
65
66          Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop⌐
            ↪   e.TempTransactionLogFilename);
67      }
68  }

69
70  [Fact]
71  public static void BasicTransactionLogTest()
72  {
73      using (var scope = new TempLinksTestScope(useLog: true))
74      {
75          var links = scope.Links;
76          var l1 = links.Create();
77          var l2 = links.Create();
78
79          Global.Trash = links.Update(l2, l2, l1, l2);
80
81          links.Delete(l1);
82
83          links.Unsync.DisposeIfPossible(); // Close links to access log
84
85          Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop⌐
            ↪   e.TempTransactionLogFilename);
86      }
87  }

88
89  [Fact]
90  public static void TransactionAutoRevertedTest()
91  {
92      // Auto Reverted (Because no commit at transaction)
93      using (var scope = new TempLinksTestScope(useLog: true))
94      {
95          var links = scope.Links;
96          var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
97          using (var transaction = transactionsLayer.BeginTransaction())
98          {
99              var l1 = links.Create();
100             var l2 = links.Create();
101
102             links.Update(l2, l2, l1, l2);
103         }
104
105         Assert.Equal(0UL, links.Count());
106
107         links.Unsync.DisposeIfPossible();
108
109         var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s⌐
            ↪   cope.TempTransactionLogFilename);
```

```
110                    Assert.Single(transitions);
111                }
112            }
113
114        [Fact]
115        public static void TransactionUserCodeErrorNoDataSavedTest()
116        {
117            // User Code Error (Autoreverted), no data saved
118            var itself = _constants.Itself;
119
120            TempLinksTestScope lastScope = null;
121            try
122            {
123                using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
                   ↪ useLog: true))
124                {
125                    var links = scope.Links;
126                    var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor⏎
                       ↪ atorBase<ulong>)links.Unsync).Links;
127                    using (var transaction = transactionsLayer.BeginTransaction())
128                    {
129                        var l1 = links.CreateAndUpdate(itself, itself);
130                        var l2 = links.CreateAndUpdate(itself, itself);
131
132                        l2 = links.Update(l2, l2, l1, l2);
133
134                        links.CreateAndUpdate(l2, itself);
135                        links.CreateAndUpdate(l2, itself);
136
137                        //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi⏎
                           ↪ tion>(scope.TempTransactionLogFilename);
138
139                        l2 = links.Update(l2, l1);
140
141                        links.Delete(l2);
142
143                        ExceptionThrower();
144
145                        transaction.Commit();
146                    }
147
148                    Global.Trash = links.Count();
149                }
150            }
151            catch
152            {
153                Assert.False(lastScope == null);
154
155                var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l⏎
                   ↪ astScope.TempTransactionLogFilename);
156
157                Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
                   ↪ transitions[0].After.IsNull());
158
159                lastScope.DeleteFiles();
160            }
161        }
162
163        [Fact]
164        public static void TransactionUserCodeErrorSomeDataSavedTest()
165        {
166            // User Code Error (Autoreverted), some data saved
167            var itself = _constants.Itself;
168
169            TempLinksTestScope lastScope = null;
170            try
171            {
172                ulong l1;
173                ulong l2;
174
175                using (var scope = new TempLinksTestScope(useLog: true))
176                {
177                    var links = scope.Links;
178                    l1 = links.CreateAndUpdate(itself, itself);
179                    l2 = links.CreateAndUpdate(itself, itself);
180
181                    l2 = links.Update(l2, l2, l1, l2);
182
183                    links.CreateAndUpdate(l2, itself);
184                    links.CreateAndUpdate(l2, itself);
```

```csharp
                    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;

            var tempDatabaseFilename = Path.GetTempFileName();
            var tempTransactionLogFilename = Path.GetTempFileName();

            // Commit
```

```
258        using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
       ↪  UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
259        using (var links = new UInt64Links(memoryAdapter))
260        {
261            using (var transaction = memoryAdapter.BeginTransaction())
262            {
263                var l1 = links.CreateAndUpdate(itself, itself);
264                var l2 = links.CreateAndUpdate(itself, itself);
265
266                Global.Trash = links.Update(l2, l2, l1, l2);
267
268                links.Delete(l1);
269
270                transaction.Commit();
271            }
272
273            Global.Trash = links.Count();
274        }
275
276        Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran⌋
       ↪  sactionLogFilename);
277
278        // Damage database
279
280        FileHelpers.WriteFirst(tempTransactionLogFilename, new
       ↪  UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
281
282        // Try load damaged database
283        try
284        {
285            // TODO: Fix
286            using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
       ↪  UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
287            using (var links = new UInt64Links(memoryAdapter))
288            {
289                Global.Trash = links.Count();
290            }
291        }
292        catch (NotSupportedException ex)
293        {
294            Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
       ↪  yet.");
295        }
296
297        Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran⌋
       ↪  sactionLogFilename);
298
299        File.Delete(tempDatabaseFilename);
300        File.Delete(tempTransactionLogFilename);
301    }
302
303    [Fact]
304    public static void Bug1Test()
305    {
306        var tempDatabaseFilename = Path.GetTempFileName();
307        var tempTransactionLogFilename = Path.GetTempFileName();
308
309        var itself = _constants.Itself;
310
311        // User Code Error (Autoreverted), some data saved
312        try
313        {
314            ulong l1;
315            ulong l2;
316
317            using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))
318            using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
       ↪  tempTransactionLogFilename))
319            using (var links = new UInt64Links(memoryAdapter))
320            {
321                l1 = links.CreateAndUpdate(itself, itself);
322                l2 = links.CreateAndUpdate(itself, itself);
323
324                l2 = links.Update(l2, l2, l1, l2);
325
326                links.CreateAndUpdate(l2, itself);
327                links.CreateAndUpdate(l2, itself);
328            }
329
```

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

331
332                using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))
333                using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
                   ↪  tempTransactionLogFilename))
334                using (var links = new UInt64Links(memoryAdapter))
335                {
336                    using (var transaction = memoryAdapter.BeginTransaction())
337                    {
338                        l2 = links.Update(l2, l1);

339
340                        links.Delete(l2);

341
342                        ExceptionThrower();

343
344                        transaction.Commit();
345                    }

346
347                    Global.Trash = links.Count();
348                }
349            }
350            catch
351            {
352                Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp↵
                   ↪  TransactionLogFilename);
353            }

354
355            File.Delete(tempDatabaseFilename);
356            File.Delete(tempTransactionLogFilename);
357        }

358
359        private static void ExceptionThrower() => throw new InvalidOperationException();

360
361        [Fact]
362        public static void PathsTest()
363        {
364            var source = _constants.SourcePart;
365            var target = _constants.TargetPart;

366
367            using (var scope = new TempLinksTestScope())
368            {
369                var links = scope.Links;
370                var l1 = links.CreatePoint();
371                var l2 = links.CreatePoint();

372
373                var r1 = links.GetByKeys(l1, source, target, source);
374                var r2 = links.CheckPathExistance(l2, l2, l2, l2);
375            }
376        }

377
378        [Fact]
379        public static void RecursiveStringFormattingTest()
380        {
381            using (var scope = new TempLinksTestScope(useSequences: true))
382            {
383                var links = scope.Links;
384                var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.

385
386                var a = links.CreatePoint();
387                var b = links.CreatePoint();
388                var c = links.CreatePoint();

389
390                var ab = links.GetOrCreate(a, b);
391                var cb = links.GetOrCreate(c, b);
392                var ac = links.GetOrCreate(a, c);

393
394                a = links.Update(a, c, b);
395                b = links.Update(b, a, c);
396                c = links.Update(c, a, b);

397
398                Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
399                Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
400                Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));

401
402                Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
                   ↪  "(5:(4:5 (6:5 4)) 6)");
403                Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
                   ↪  "(6:(5:(4:5 6) 6) 4)");
```

```csharp
                    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;
```

```
477            var totalLinksToCreate = gibibyte /
    ↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
478            var linksStep = 102 * mebibyte /
    ↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
479
480            var creationMeasurements = new List<TimeSpan>();
481            var searchMeasuremets = new List<TimeSpan>();
482            var deletionMeasurements = new List<TimeSpan>();
483
484            GetBaseRandomLoopOverhead(linksStep);
485            GetBaseRandomLoopOverhead(linksStep);
486
487            var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
488
489            ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
490
491            var loops = totalLinksToCreate / linksStep;
492
493            for (int i = 0; i < loops; i++)
494            {
495                creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
496                searchMeasuremets.Add(Measure(() => links.RunRandomSearches(linksStep)));
497
498                Console.Write("\rC + S {0}/{1}", i + 1, loops);
499            }
500
501            ConsoleHelpers.Debug();
502
503            for (int i = 0; i < loops; i++)
504            {
505                deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
506
507                Console.Write("\rD {0}/{1}", i + 1, loops);
508            }
509
510            ConsoleHelpers.Debug();
511
512            ConsoleHelpers.Debug("C S D");
513
514            for (int i = 0; i < loops; i++)
515            {
516                ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
    ↪ searchMeasuremets[i], deletionMeasurements[i]);
517            }
518
519            ConsoleHelpers.Debug("C S D (no overhead)");
520
521            for (int i = 0; i < loops; i++)
522            {
523                ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] - stepLoopOverhead,
    ↪ searchMeasuremets[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
524            }
525
526            ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
    ↪ links.Total);
527        }
528
529        private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
    ↪ amountToCreate)
530        {
531            for (long i = 0; i < amountToCreate; i++)
532                links.Create(0, 0);
533        }
534
535         private static TimeSpan GetBaseRandomLoopOverhead(long loops)
536         {
537             return Measure(() =>
538             {
539                 ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
540                 ulong result = 0;
541                 for (long i = 0; i < loops; i++)
542                 {
543                     var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
544                     var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
545
546                     result += maxValue + source + target;
547                 }
548                 Global.Trash = result;
549             });
```

```csharp
        }
         */

        [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()
        {
```

```csharp
            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;
```

```csharp
                    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();
```

```csharp
                    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()
        {
```

```csharp
                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
    }
}
```

```csharp
using Platform.Data.Doublets.Memory;
using Platform.Data.Doublets.Memory.United.Generic;
using Platform.Data.Numbers.Raw;
using Platform.Memory;
using Platform.Numbers;
using Xunit;
using Xunit.Abstractions;
using TLink = System.UInt64;

namespace Platform.Data.Doublets.Tests
{
    public class Uint64LinksExtensionsTests
    {
        public static ILinks<TLink> CreateLinks() => CreateLinks<TLink>(new
        ↪   Platform.IO.TemporaryFile());

        public static ILinks<TLink> CreateLinks<TLink>(string dataDBFilename)
        {
            var linksConstants = new LinksConstants<TLink>(enableExternalReferencesSupport:
            ↪   true);
            return new UnitedMemoryLinks<TLink>(new
            ↪   FileMappedResizableDirectMemory(dataDBFilename),
            ↪   UnitedMemoryLinks<TLink>.DefaultLinksSizeStep, linksConstants,
            ↪   IndexTreeType.Default);
        }
        [Fact]
        public void FormatStructureWithExternalReferenceTest()
        {
            ILinks<TLink> links = CreateLinks();
            TLink zero = default;
            var one = Arithmetic.Increment(zero);
            var markerIndex = one;
            var meaningRoot = links.GetOrCreate(markerIndex, markerIndex);
            var numberMarker = links.GetOrCreate(meaningRoot, Arithmetic.Increment(ref
            ↪   markerIndex));
            AddressToRawNumberConverter<TLink> addressToNumberConverter = new();
            var numberAddress = addressToNumberConverter.Convert(1);
            var numberLink = links.GetOrCreate(numberMarker, numberAddress);
            var linkNotation = links.FormatStructure(numberLink, link => link.IsFullPoint(),
            ↪   true);
            Assert.Equal("(3:(2:1 2) 18446744073709551615)", linkNotation);
        }
    }
}
```

```csharp
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);
```

```
29              var fromUnaryNumberConverterUsingAddOperation = new
    ↪   UnaryNumberToAddressAddOperationConverter<ulong>(links, one);
30              for (int i = 0; i < N; i++)
31              {
32                  Assert.Equal(numbers[i],
    ↪   fromUnaryNumberConverterUsingOrOperation.Convert(unaryNumbers[i]));
33                  Assert.Equal(numbers[i],
    ↪   fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));
34              }
35          }
36      }
37  }
38 }
```

## 1.188  ./csharp/Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs

```
1   using Xunit;
2   using Platform.Converters;
3   using Platform.Memory;
4   using Platform.Reflection;
5   using Platform.Scopes;
6   using Platform.Data.Numbers.Raw;
7   using Platform.Data.Doublets.Incrementers;
8   using Platform.Data.Doublets.Numbers.Unary;
9   using Platform.Data.Doublets.PropertyOperators;
10  using Platform.Data.Doublets.Sequences.Converters;
11  using Platform.Data.Doublets.Sequences.Indexes;
12  using Platform.Data.Doublets.Sequences.Walkers;
13  using Platform.Data.Doublets.Unicode;
14  using Platform.Data.Doublets.Memory.United.Generic;
15  using Platform.Data.Doublets.CriterionMatchers;
16
17  namespace Platform.Data.Doublets.Tests
18  {
19      public static class UnicodeConvertersTests
20      {
21          [Fact]
22          public static void CharAndUnaryNumberUnicodeSymbolConvertersTest()
23          {
24              using (var scope = new TempLinksTestScope())
25              {
26                  var links = scope.Links;
27                  var meaningRoot = links.CreatePoint();
28                  var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
29                  var powerOf2ToUnaryNumberConverter = new
    ↪   PowerOf2ToUnaryNumberConverter<ulong>(links, one);
30                  var addressToUnaryNumberConverter = new
    ↪   AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
31                  var unaryNumberToAddressConverter = new
    ↪   UnaryNumberToAddressOrOperationConverter<ulong>(links,
    ↪   powerOf2ToUnaryNumberConverter);
32                  TestCharAndUnicodeSymbolConverters(links, meaningRoot,
    ↪   addressToUnaryNumberConverter, unaryNumberToAddressConverter);
33              }
34          }
35
36          [Fact]
37          public static void CharAndRawNumberUnicodeSymbolConvertersTest()
38          {
39              using (var scope = new Scope<Types<HeapResizableDirectMemory,
    ↪   UnitedMemoryLinks<ulong>>>())
40              {
41                  var links = scope.Use<ILinks<ulong>>();
42                  var meaningRoot = links.CreatePoint();
43                  var addressToRawNumberConverter = new AddressToRawNumberConverter<ulong>();
44                  var rawNumberToAddressConverter = new RawNumberToAddressConverter<ulong>();
45                  TestCharAndUnicodeSymbolConverters(links, meaningRoot,
    ↪   addressToRawNumberConverter, rawNumberToAddressConverter);
46              }
47          }
48
49          private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
    ↪   meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
    ↪   numberToAddressConverter)
50          {
51              var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
52              var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links,
    ↪   addressToNumberConverter, unicodeSymbolMarker);
53              var originalCharacter = 'H';
54              var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);
```

```csharp
                    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);
```

```
105                    var resultingString =
                       ↪  unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
106
107                    Assert.Equal(originalString, resultingString);
108                }
109            }
110        }
111    }
```

## 1.189 ./csharp/Platform.Data.Doublets.Tests/UnitedMemoryUInt32LinksTests.cs

```
1    using System;
2    using Xunit;
3    using Platform.Reflection;
4    using Platform.Memory;
5    using Platform.Scopes;
6    using Platform.Data.Doublets.Memory.United.Specific;
7    using TLink = System.UInt32;
8
9    namespace Platform.Data.Doublets.Tests
10   {
11       public unsafe static class UnitedMemoryUInt32LinksTests
12       {
13           [Fact]
14           public static void CRUDTest()
15           {
16               Using(links => links.TestCRUDOperations());
17           }
18
19           [Fact]
20           public static void RawNumbersCRUDTest()
21           {
22               Using(links => links.TestRawNumbersCRUDOperations());
23           }
24
25           [Fact]
26           public static void MultipleRandomCreationsAndDeletionsTest()
27           {
28               Using(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultip⏎
                 ↪  leRandomCreationsAndDeletions(100));
29           }
30
31           private static void Using(Action<ILinks<TLink>> action)
32           {
33               using (var scope = new Scope<Types<HeapResizableDirectMemory,
                 ↪  UInt32UnitedMemoryLinks>>())
34               {
35                   action(scope.Use<ILinks<TLink>>());
36               }
37           }
38       }
39   }
```

## 1.190 ./csharp/Platform.Data.Doublets.Tests/UnitedMemoryUInt64LinksTests.cs

```
1    using System;
2    using Xunit;
3    using Platform.Reflection;
4    using Platform.Memory;
5    using Platform.Scopes;
6    using Platform.Data.Doublets.Memory.United.Specific;
7    using TLink = System.UInt64;
8
9    namespace Platform.Data.Doublets.Tests
10   {
11       public unsafe static class UnitedMemoryUInt64LinksTests
12       {
13           [Fact]
14           public static void CRUDTest()
15           {
16               Using(links => links.TestCRUDOperations());
17           }
18
19           [Fact]
20           public static void RawNumbersCRUDTest()
21           {
22               Using(links => links.TestRawNumbersCRUDOperations());
23           }
24
25           [Fact]
26           public static void MultipleRandomCreationsAndDeletionsTest()
```

```csharp
            {
                Using(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultip
                ↪  leRandomCreationsAndDeletions(100));
            }

        private static void Using(Action<ILinks<TLink>> action)
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪  UInt64UnitedMemoryLinks>>())
            {
                action(scope.Use<ILinks<TLink>>());
            }
        }
    }
}
```

# Index