LinksPlatform's Platform.Data.Doublets Class Library

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

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

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

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

        private readonly TLink _targetToMatch;

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

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

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

```csharp
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

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

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

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

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

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

        protected ILinks<TLink> _facade;

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

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

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

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

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

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

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

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

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

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

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

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

        protected readonly DisposableWithMultipleCallsAllowed Disposable;

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

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

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

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

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

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

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

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

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

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

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

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

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

```

```
#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

```
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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
using System.Collections.Generic;
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.Decorators
{
    public class UInt32Links : LinksDisposableDecoratorBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32Links(ILinks<TLink> links) : base(links) { }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
        {
            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<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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
            ////            Func<T, bool> handler = link =>
            ////            {
            ////                var matchedLink = Memory.GetLinkValue(link);
            ////                var matchDecision = matchedHandler(matchedLink, matchedLink);
            ////                return !Equals(matchDecision, Constants.Break);
            ////            };
            ////            if (!Memory.Each(handler, restriction))
            ////                return Constants.Break;
            ////        }
            ////    }
            ////    else
            ////    {
            ////        if (substitution != null)
            ////        {
            ////            transitions = new List<IList<T>>();
            ////            Func<T, bool> handler = link =>
            ////            {
            ////                var matchedLink = Memory.GetLinkValue(link);
            ////                transitions.Add(matchedLink);
            ////                return true;
            ////            };
            ////            if (!Memory.Each(handler, restriction))
            ////                return Constants.Break;
            ////        }
            ////        else
            ////        {
            ////            return Constants.Continue;
            ////        }
            ////    }
            ////}
            ////if (substitution != null)
            ////{
            ////    // Есть причина делать замену (запись)
            ////    if (substitutedHandler != null)
            ////    {
            ////    }
            ////    else
            ////    {
            ////    }
            ////}
            ////return Constants.Continue;

            //if (restriction.IsNullOrEmpty()) // Create
            //{
            //    substitution[Constants.IndexPart] = Memory.AllocateLink();
            //    Memory.SetLinkValue(substitution);
            //}
            //else if (substitution.IsNullOrEmpty()) // Delete
            //{
            //    Memory.FreeLink(restriction[Constants.IndexPart]);
            //}
            //else if (restriction.EqualTo(substitution)) // Read or ("repeat" the state) // Each
            //{
            //    // No need to collect links to list
            //    // Skip == Continue
            //    // No need to check substituedHandler
            //    if (!Memory.Each(link => !Equals(matchedHandler(Memory.GetLinkValue(link)),
            ↪  Constants.Break), restriction))
            //        return Constants.Break;
            //}
            //else // Update
            //{
            //    //List<IList<T>> matchedLinks = null;
            //    if (matchedHandler != null)
            //    {
            //        matchedLinks = new List<IList<T>>();
            //        Func<T, bool> handler = link =>
            //        {
            //            var matchedLink = Memory.GetLinkValue(link);
            //            var matchDecision = matchedHandler(matchedLink);
            //            if (Equals(matchDecision, Constants.Break))
            //                return false;
            //            if (!Equals(matchDecision, Constants.Skip))
            //                matchedLinks.Add(matchedLink);
            //            return true;
            //        };
            //        if (!Memory.Each(handler, restriction))
            //            return Constants.Break;
            //    }
            //    if (!matchedLinks.IsNullOrEmpty())
            //    {
            //        var totalMatchedLinks = matchedLinks.Count;
            //        for (var i = 0; i < totalMatchedLinks; i++)
            //        {
            //            var matchedLink = matchedLinks[i];
            //            if (substitutedHandler != null)
            //            {
            //                var newValue = new List<T>(); // TODO: Prepare value to update here
            //                // TODO: Decide is it actually needed to use Before and After substitution handling.
            //                var substitutedDecision = substitutedHandler(matchedLink, newValue);
            //                if (Equals(substitutedDecision, Constants.Break))
            //                    return Constants.Break;
            //                if (Equals(substitutedDecision, Constants.Continue))
            //                {
            //                    // Actual update here
            //                    Memory.SetLinkValue(newValue);
            //                }
            //                if (Equals(substitutedDecision, Constants.Skip))
            //                {
            //                    // Cancel the update. TODO: decide use separate Cancel constant or Skip is enough?
            //                }
            //            }
            //        }
            //    }
            //}
            return _constants.Continue;
        }

        public TLink Trigger(IList<TLink> patternOrCondition, Func<IList<TLink>, TLink> matchHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink> substitutionHandler)
        {
            var constants = _constants;
            if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
            {
                return constants.Continue;
            }
            else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO: Check if it is a correct condition
            {
                // Or it only applies to trigger without matchHandler.
                throw new NotImplementedException();
            }
            else if (!substitution.IsNullOrEmpty()) // Creation
            {
                var before = Array.Empty<TLink>();
                // Что должно означать False здесь? Остановиться (перестать идти) или пропустить (пройти мимо) или пустить (взять)?
                if (matchHandler != null && _equalityComparer.Equals(matchHandler(before), constants.Break))
                {
                    return constants.Break;
                }
                var after = (IList<TLink>)substitution.ToArray();
                if (_equalityComparer.Equals(after[0], default))
                {
                    var newLink = _links.Create();
                    after[0] = newLink;
                }
                if (substitution.Count == 1)
                {
                    after = _links.GetLink(substitution[0]);
                }
                else if (substitution.Count == 3)
                {
                    //Links.Create(after);
                }
                else
                {
                    throw new NotSupportedException();
                }
                if (matchHandler != null)
                {
                    return substitutionHandler(before, after);
                }
                return constants.Continue;
            }
            else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
            {
                if (patternOrCondition.Count == 1)
                {
                    var linkToDelete = patternOrCondition[0];
                    var before = _links.GetLink(linkToDelete);
                    if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                    ↪  constants.Break))
                    {
                        return constants.Break;
                    }
                    var after = Array.Empty<TLink>();
                    _links.Update(linkToDelete, constants.Null, constants.Null);
                    _links.Delete(linkToDelete);
                    if (matchHandler != null)
                    {
                        return substitutionHandler(before, after);
                    }
                    return constants.Continue;
                }
                else
                {
                    throw new NotSupportedException();
                }
            }
            else // Replace / Update
            {
                if (patternOrCondition.Count == 1) //-V3125
                {
                    var linkToUpdate = patternOrCondition[0];
                    var before = _links.GetLink(linkToUpdate);
                    if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                    ↪  constants.Break))
                    {
                        return constants.Break;
                    }
                    var after = (IList<TLink>)substitution.ToArray(); //-V3125
                    if (_equalityComparer.Equals(after[0], default))
                    {
                        after[0] = linkToUpdate;
                    }
                    if (substitution.Count == 1)
                    {
                        if (!_equalityComparer.Equals(substitution[0], linkToUpdate))
                        {
                            after = _links.GetLink(substitution[0]);
                            _links.Update(linkToUpdate, constants.Null, constants.Null);
                            _links.Delete(linkToUpdate);
                        }
                    }
                    else if (substitution.Count == 3)
                    {
                        //Links.Update(after);
                    }
                    else
                    {
                        throw new NotSupportedException();
                    }
                    if (matchHandler != null)
                    {
                        return substitutionHandler(before, after);
                    }
                    return constants.Continue;
                }
                else
                {
                    throw new NotSupportedException();
                }
            }
        }

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

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

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

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

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

namespace Platform.Data.Doublets
{

    /// <summury>
    /// <para>.</para>
    /// <para>.</para>
    /// </summury>
    /// <typeparam>
    /// <para>.</para>
    /// <para>.</para>
    /// </typeparam>
    public struct Doublet<T> : IEquatable<Doublet<T>>
    {
        private static readonly EqualityComparer<T> _equalityComparer =
        ↪  EqualityComparer<T>.Default;

        /// <summury>
        /// <para>.</para>
        /// <para>.</para>
        /// </summury>
        /// <typeparam name="T">
        /// <para>.</para>
        /// <para>.</para>
        /// </typeparam>
        public readonly T Source;

        /// <summury>
        /// <para>.</para>
        /// <para>.</para>
        /// </summury>
        /// <typeparam name="T">
        /// <para>.</para>
        /// <para>.</para>
        /// </typeparam>
        public readonly T Target;

        /// <summury>
        /// <para>.</para>
        /// <para>.</para>
        /// </summury>
        /// <typeparam name="T">
        /// <para>.</para>
        /// <para>.</para>
        /// </typeparam>
        /// <param name="source">
        /// <para>.</para>
        /// <para>.</para>
        /// </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>
        /// <param name="left">
        /// <para>.</para>
        /// <para>.</para>
        /// </param>
        /// <param name="right">
        /// <para>.</para>
        /// <para>.</para>
        /// </param>
        /// <returns>
        /// <para>.</para>
        /// <para>.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator ==(Doublet<T> left, Doublet<T> right) => left.Equals(right);

        /// <summury>
        /// <para>.</para>
        /// <para>.</para>
        /// </summury>
        /// <param name="left">
        /// <para>.</para>
        /// <para>.</para>
        /// </param>
        /// <param name="right">
        /// <para>.</para>
        /// <para>.</para>
        /// </param>
        /// <returns>
        /// <para>.</para>
        /// <para>.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator !=(Doublet<T> left, Doublet<T> right) => !(left == right);
    }
}
```

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

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

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

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

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

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

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

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

using System.Collections.Generic;

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

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

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

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

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

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

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

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

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

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

        [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
        /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
        /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[]
        ↪  path)
        {
            var current = path[0];
            //EnsureLinkExists(current, "path");
            if (!links.Exists(current))
            {
                return false;
            }
            var equalityComparer = EqualityComparer<TLink>.Default;
            var constants = links.Constants;
            for (var i = 1; i < path.Length; i++)
            {
                var next = path[i];
                var values = links.GetLink(current);
                var source = values[constants.SourcePart];
                var target = values[constants.TargetPart];
                if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
                ↪  next))
                {
                    //throw new InvalidOperationException(string.Format("Невозможно выбрать
                    ↪  путь, так как и Source и Target совпадают с элементом пути {0}.", next));
                    return false;
                }
                if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
                ↪  target))
                {
                    //throw new InvalidOperationException(string.Format("Невозможно продолжить
                    ↪  путь через элемент пути {0}", next));
                    return false;
                }
                current = next;
            }
            return true;
        }

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

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

        #endregion

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
        ↪  restrictions)
        {
            var arraySize = CheckedConverter<TLink,
            ↪  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>();
            }
        }
        /// <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.Maximum)));
                var createdLinks = new List<TLink>();
                var equalityComparer = EqualityComparer<TLink>.Default;
                TLink createdLink = creator();
                while (!equalityComparer.Equals(createdLink, max))
                {
                    createdLinks.Add(createdLink);
                }
                for (var i = 0; i < createdLinks.Count; i++)
                {
                    if (!nonExistentAddresses.Contains(createdLinks[i]))
                    {
                        links.Delete(createdLinks[i]);
                    }
                }
            }
        }

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

        /// <param name="links">Хранилище связей.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool HasUsages<TLink>(this ILinks<TLink> links, TLink link) =>
         ↪  Comparer<TLink>.Default.Compare(links.CountUsages(link), default) > 0;

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

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

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

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

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

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

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

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

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

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

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

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

        /// <remarks>Before execution of this method ensure that deleted link is detached (all
        ↪  values - source and target are reset to null) or it might enter into infinite
        ↪  recursion.</remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
        {
            var anyConstant = links.Constants.Any;
            var usagesAsSourceQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
            links.DeleteByQuery(usagesAsSourceQuery);
            var usagesAsTargetQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
            links.DeleteByQuery(usagesAsTargetQuery);
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void DeleteByQuery<TLink>(this ILinks<TLink> links, Link<TLink> query)
        {
            var count = CheckedConverter<TLink, long>.Default.Convert(links.Count(query));
            if (count > 0)
            {
                var queryResult = new TLink[count];
                var queryResultFiller = new ArrayFiller<TLink, TLink>(queryResult,
                 ↪  links.Constants.Continue);
                links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
                for (var i = count - 1; i >= 0; i--)
                {
                    links.Delete(queryResult[i]);
                }
            }
        }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        private static readonly LinksConstants<TLink> _constants =
         ↪  Default<LinksConstants<TLink>>.Instance;
        private static readonly EqualityComparer<TLink> _equalityComparer =
         ↪  EqualityComparer<TLink>.Default;

        private const int Length = 3;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        #endregion
    }
}
```

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

```
using System.Runtime.CompilerServices;

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

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

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

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

```
using System.Runtime.CompilerServices;

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

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

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

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

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

```
using System.Runtime.CompilerServices;

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

```

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

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

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

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

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

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

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

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

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

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

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

```
#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
    }
}
```

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

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

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

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

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

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

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

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

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

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

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

1.29 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksRecursionlessSizeBalancedTreeMethod

```
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);

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

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

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

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

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

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

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

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

        /// <summary>
        /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
         ↪  (концом).
        /// </summary>
        /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
        /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
        /// <returns>Индекс искомой связи.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Search(TLink source, TLink target)
        {
            var root = GetTreeRoot();
            while (!EqualToZero(root))
            {
                ref var rootLink = ref GetLinkDataPartReference(root);
                var rootSource = rootLink.Source;
                var rootTarget = rootLink.Target;
                if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                ↪  node.Key < root.Key
                {
                    root = GetLeftOrDefault(root);
                }
                else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
                ↪  node.Key > root.Key
                {
                    root = GetRightOrDefault(root);
                }
                else // node.Key == root.Key
                {
                    return root;
                }
            }
            return Zero;
        }

        // TODO: Return indices range instead of references count
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink CountUsages(TLink link)
        {
            var root = GetTreeRoot();
            var total = GetSize(root);
            var totalRightIgnore = Zero;
            while (!EqualToZero(root))
            {
                var @base = GetBasePartValue(root);
                if (LessOrEqualThan(@base, link))
                {
                    root = GetRightOrDefault(root);
                }
                else
                {
                    totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
                    root = GetLeftOrDefault(root);
                }
            }
            root = GetTreeRoot();
            var totalLeftIgnore = Zero;
            while (!EqualToZero(root))
            {
                var @base = GetBasePartValue(root);
                if (GreaterOrEqualThan(@base, link))
                {
                    root = GetLeftOrDefault(root);
                }
                else
                {
                    totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
                    root = GetRightOrDefault(root);
                }
            }
            return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
        }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        /// <summary>
        /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target (концом).
        /// </summary>
        /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
        /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
        /// <returns>Индекс искомой связи.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Search(TLink source, TLink target)
        {
            var root = GetTreeRoot();
            while (!EqualToZero(root))
            {
                ref var rootLink = ref GetLinkDataPartReference(root);
                var rootSource = rootLink.Source;
                var rootTarget = rootLink.Target;
                if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) // node.Key < root.Key
                {
                    root = GetLeftOrDefault(root);
                }
                else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) // node.Key > root.Key
                {
                    root = GetRightOrDefault(root);
                }
                else // node.Key == root.Key
                {
                    return root;
                }
            }
            return Zero;
        }

        // TODO: Return indices range instead of references count
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink CountUsages(TLink link)
        {
            var root = GetTreeRoot();
            var total = GetSize(root);
            var totalRightIgnore = Zero;
            while (!EqualToZero(root))
            {
                var @base = GetBasePartValue(root);
                if (LessOrEqualThan(@base, link))
                {
                    root = GetRightOrDefault(root);
                }
                else
                {
                    totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
                    root = GetLeftOrDefault(root);
                }
            }
            root = GetTreeRoot();
            var totalLeftIgnore = Zero;
            while (!EqualToZero(root))
            {
                var @base = GetBasePartValue(root);
                if (GreaterOrEqualThan(@base, link))
                {
                    root = GetLeftOrDefault(root);
                }
                else
                {
                    totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
                    root = GetRightOrDefault(root);
                }
            }
            return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
        ↪  EachUsageCore(@base, GetTreeRoot(), handler);

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

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

1.31 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSourcesRecursionlessSizeBalancedTree

```
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 ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods<TLink> :
    ↪  ExternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }
    }
}
```

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

```
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.33 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksTargetsRecursionlessSizeBalancedTree

```
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)]
        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;
        }
    }
}
```

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

```
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.35 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksRecursionlessSizeBalancedTreeMethod

```
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;
        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>
        /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
         ↪  (концом).
        /// </summary>
        /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
        /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
        /// <returns>Индекс искомой связи.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public abstract TLink Search(TLink source, TLink target);

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        /// <summary>
        /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
        ↪  (концом).
        /// </summary>
        /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
        /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
        /// <returns>Индекс искомой связи.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public abstract TLink Search(TLink source, TLink target);

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

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

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

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

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

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

```
using System;
using System.Collections.Generic;
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 InternalLinksSourcesLinkedListMethods<TLink> :
     ↪  RelativeCircularDoublyLinkedListMethods<TLink>
    {
        private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
         ↪  UncheckedConverter<TLink, long>.Default;
        private readonly byte* _linksDataParts;
        private readonly byte* _linksIndexParts;
        protected readonly TLink Break;
        protected readonly TLink Continue;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public InternalLinksSourcesLinkedListMethods(LinksConstants<TLink> constants, byte*
         ↪  linksDataParts, byte* linksIndexParts)
        {
            _linksDataParts = linksDataParts;
            _linksIndexParts = linksIndexParts;
            Break = constants.Break;
            Continue = constants.Continue;
        }

        [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 TLink GetFirst(TLink head) =>
         ↪  GetLinkIndexPartReference(head).RootAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetLast(TLink head)
        {
            var first = GetLinkIndexPartReference(head).RootAsSource;
            if (EqualToZero(first))
            {
                return first;
            }
            else
            {
                return GetPrevious(first);
            }
        }

        [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;
        }
    }
}
```

1.38 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSourcesRecursionlessSizeBalancedTree

```
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);
    }
}
```

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

```
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.40 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksTargetsRecursionlessSizeBalancedTree

```
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>
    {
        [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);
    }
}
```

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

```
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SplitMemoryLinks(string dataMemory, string indexMemory) : this(new
         ↪  FileMappedResizableDirectMemory(dataMemory), new
         ↪  FileMappedResizableDirectMemory(indexMemory)) { }

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SplitMemoryLinks(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
                ↪   InternalLinksSourcesSizeBalancedTreeMethods<TLink>(Constants,
                ↪   _linksDataParts, _linksIndexParts, _header);
                _createExternalSourceTreeMethods = () => new
                ↪   ExternalLinksSourcesSizeBalancedTreeMethods<TLink>(Constants,
                ↪   _linksDataParts, _linksIndexParts, _header);
                _createInternalTargetTreeMethods = () => new
                ↪   InternalLinksTargetsSizeBalancedTreeMethods<TLink>(Constants,
                ↪   _linksDataParts, _linksIndexParts, _header);
                _createExternalTargetTreeMethods = () => new
                ↪   ExternalLinksTargetsSizeBalancedTreeMethods<TLink>(Constants,
                ↪   _linksDataParts, _linksIndexParts, _header);
            }
            else
            {
                _createInternalSourceTreeMethods = () => new
                ↪   InternalLinksSourcesRecursionlessSizeBalancedTreeMethods<TLink>(Constants,
                ↪   _linksDataParts, _linksIndexParts, _header);
                _createExternalSourceTreeMethods = () => new
                ↪   ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods<TLink>(Constants,
                ↪   _linksDataParts, _linksIndexParts, _header);
                _createInternalTargetTreeMethods = () => new
                ↪   InternalLinksTargetsRecursionlessSizeBalancedTreeMethods<TLink>(Constants,
                ↪   _linksDataParts, _linksIndexParts, _header);
                _createExternalTargetTreeMethods = () => new
                ↪   ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods<TLink>(Constants,
                ↪   _linksDataParts, _linksIndexParts, _header);
            }
            Init(dataMemory, indexMemory);
        }

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

```csharp

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
         ↪  indexMemory, long memoryReservationStep, LinksConstants<TLink> constants, 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
            if (minimumDataReservedCapacity % _dataMemoryReservationStepInBytes > 0)
            {
                minimumDataReservedCapacity = ((minimumDataReservedCapacity /
                 ↪  _dataMemoryReservationStepInBytes) * _dataMemoryReservationStepInBytes) +
                 ↪  _dataMemoryReservationStepInBytes;
            }
            if (minimumIndexReservedCapacity % _indexMemoryReservationStepInBytes > 0)
            {
                minimumIndexReservedCapacity = ((minimumIndexReservedCapacity /
                 ↪  _indexMemoryReservationStepInBytes) * _indexMemoryReservationStepInBytes) +
                 ↪  _indexMemoryReservationStepInBytes;
            }
            if (dataMemory.ReservedCapacity != minimumDataReservedCapacity)
            {
                dataMemory.ReservedCapacity = minimumDataReservedCapacity;
            }
            if (indexMemory.ReservedCapacity != minimumIndexReservedCapacity)
            {
                indexMemory.ReservedCapacity = minimumIndexReservedCapacity;
            }
            SetPointers(dataMemory, indexMemory);
            header = ref GetHeaderReference();
            // Ensure correctness _memory.UsedCapacity over _header->AllocatedLinks
            // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
            dataMemory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) *
             ↪  LinkDataPartSizeInBytes) + LinkDataPartSizeInBytes; // First link is read only
             ↪  zero link.
            indexMemory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) *
             ↪  LinkIndexPartSizeInBytes) + LinkHeaderSizeInBytes;
            // Ensure correctness _memory.ReservedLinks over _header->ReservedCapacity
            // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
            header.ReservedLinks = ConvertToAddress((dataMemory.ReservedCapacity -
             ↪  LinkDataPartSizeInBytes) / LinkDataPartSizeInBytes);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Count(IList<TLink> restrictions)
        {
            // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
            if (restrictions.Count == 0)
            {
                return Total;
            }
            var constants = Constants;
            var any = constants.Any;
            var index = restrictions[constants.IndexPart];
            if (restrictions.Count == 1)
            {
                if (AreEqual(index, any))
                {
                    return Total;
                }
                return Exists(index) ? GetOne() : GetZero();
            }
            if (restrictions.Count == 2)
            {
                var value = restrictions[1];
                if (AreEqual(index, any))
                {
                    if (AreEqual(value, any))
                    {
                        return Total; // Any - как отсутствие ограничения
                    }
                    var externalReferencesRange = constants.ExternalReferencesRange;
                    if (externalReferencesRange.HasValue &&
                     ↪  externalReferencesRange.Value.Contains(value))
                    {
                        return Add(ExternalSourcesTreeMethods.CountUsages(value),
                         ↪  ExternalTargetsTreeMethods.CountUsages(value));
                    }
                    else
                    {
                        if (_useLinkedList)
                        {
                            return Add(InternalSourcesListMethods.CountUsages(value),
                             ↪  InternalTargetsTreeMethods.CountUsages(value));
                        }
                        else
                        {
                            return Add(InternalSourcesTreeMethods.CountUsages(value),
                             ↪  InternalTargetsTreeMethods.CountUsages(value));
                        }
                    }
                }
                else
                {
                    if (!Exists(index))
                    {
                        return GetZero();
                    }
                    if (AreEqual(value, any))
                    {
                        return GetOne();
                    }
                    ref var storedLinkValue = ref GetLinkDataPartReference(index);
                    if (AreEqual(storedLinkValue.Source, value) ||
                    ↪  AreEqual(storedLinkValue.Target, value))
                    {
                        return GetOne();
                    }
                    return GetZero();
                }
            }
            if (restrictions.Count == 3)
            {
                var externalReferencesRange = constants.ExternalReferencesRange;
                var source = restrictions[constants.SourcePart];
                var target = restrictions[constants.TargetPart];
                if (AreEqual(index, any))
                {
                    if (AreEqual(source, any) && AreEqual(target, any))
                    {
                        return Total;
                    }
                    else if (AreEqual(source, any))
                    {
                        if (externalReferencesRange.HasValue &&
                        ↪  externalReferencesRange.Value.Contains(target))
                        {
                            return ExternalTargetsTreeMethods.CountUsages(target);
                        }
                        else
                        {
                            return InternalTargetsTreeMethods.CountUsages(target);
                        }
                    }
                    else if (AreEqual(target, any))
                    {
                        if (externalReferencesRange.HasValue &&
                        ↪  externalReferencesRange.Value.Contains(source))
                        {
                            return ExternalSourcesTreeMethods.CountUsages(source);
                        }
                        else
                        {
                            if (_useLinkedList)
                            {
                                return InternalSourcesListMethods.CountUsages(source);
                            }
                            else
                            {
                                return InternalSourcesTreeMethods.CountUsages(source);
                            }
                        }
                    }
                    else //if(source != Any && target != Any)
                    {
                        // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
                        TLink link;
                        if (externalReferencesRange.HasValue)
                        {
                            if (externalReferencesRange.Value.Contains(source) &&
                            ↪  externalReferencesRange.Value.Contains(target))
                            {
                                link = ExternalSourcesTreeMethods.Search(source, target);
                            }
                            else if (externalReferencesRange.Value.Contains(source))
                            {
                                link = InternalTargetsTreeMethods.Search(source, target);
                            }
                            else if (externalReferencesRange.Value.Contains(target))
                            {
                                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();
                    }
                    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
                    {
                        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))
            {
                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;
            if (!AreEqual(freeLink, Constants.Null))
            {
                UnusedLinksListMethods.Detach(freeLink);
            }
            else
            {
                var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
                if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
                {
                    throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
                }
                if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
                {
                    _dataMemory.ReservedCapacity += _dataMemoryReservationStepInBytes;
                    _indexMemory.ReservedCapacity += _indexMemoryReservationStepInBytes;
                    SetPointers(_dataMemory, _indexMemory);
                    header = ref GetHeaderReference();
                    header.ReservedLinks = ConvertToAddress(_dataMemory.ReservedCapacity /
                     ↪  LinkDataPartSizeInBytes);
                }
                freeLink = header.AllocatedLinks = Increment(header.AllocatedLinks);
                _dataMemory.UsedCapacity += LinkDataPartSizeInBytes;
                _indexMemory.UsedCapacity += LinkIndexPartSizeInBytes;
            }
            return freeLink;
        }

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

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

        /// <remarks>
        /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
         ↪  адрес реально поменялся
        ///
        /// Указатель this.links может быть в том же месте,
        /// так как 0-я связь не используется и имеет такой же размер как Header,
        /// поэтому header размещается в том же месте, что и 0-я связь
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetPointers(IResizableDirectMemory dataMemory,
         ↪  IResizableDirectMemory indexMemory);

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

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

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

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

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

        [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
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Memory.Split.Generic
{
    public unsafe class UnusedLinksListMethods<TLink> :
     ↪  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)]
        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;
    }
}
```

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

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

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

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

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

        public TLink Source;
        public TLink Target;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.47 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksRecursionlessSizeBalancedTree

```
using System.Runtime.CompilerServices;
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 abstract class UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase :
    ↪  ExternalLinksRecursionlessSizeBalancedTreeMethodsBase<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
        ↪  UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase(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)
        {
            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);
        }
    }
}
```

1.48 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksSizeBalancedTreeMethodsBase

```
using System.Runtime.CompilerServices;
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 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)
        {
            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);
        }
    }
}
```

1.49 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksSourcesRecursionlessSizeBalan

```
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)]
        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;
        }
    }
}
```

1.50 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksSourcesSizeBalancedTreeMeth

```
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 UInt32ExternalLinksSourcesSizeBalancedTreeMethods :
     ↪  UInt32ExternalLinksSizeBalancedTreeMethodsBase
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32ExternalLinksSourcesSizeBalancedTreeMethods(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)]
        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;
            }
        }
    }
```

1.51 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksTargetsRecursionlessSizeBalanc

```
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 UInt32ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods :
     ↪  UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public
         ↪  UInt32ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(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;

        [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() => Header->RootAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }
    }
}
```

1.52 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksTargetsSizeBalancedTreeMetho

```
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 UInt32ExternalLinksTargetsSizeBalancedTreeMethods :
     ↪  UInt32ExternalLinksSizeBalancedTreeMethodsBase
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32ExternalLinksTargetsSizeBalancedTreeMethods(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;

        [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() => Header->RootAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }
    }
}
```

1.53 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksRecursionlessSizeBalancedTreeM

```
using System.Runtime.CompilerServices;
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 abstract class UInt32InternalLinksRecursionlessSizeBalancedTreeMethodsBase :
     ↪  InternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
    {
        protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
        protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
        protected new readonly LinksHeader<TLink>* Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected
         ↪  UInt32InternalLinksRecursionlessSizeBalancedTreeMethodsBase(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 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);
    }
}
```

1.54 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksSizeBalancedTreeMethodsBase.

```
using System.Runtime.CompilerServices;
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 abstract class UInt32InternalLinksSizeBalancedTreeMethodsBase :
    ↪  InternalLinksSizeBalancedTreeMethodsBase<TLink>
    {
        protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
        protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
        protected new readonly LinksHeader<TLink>* Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected UInt32InternalLinksSizeBalancedTreeMethodsBase(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 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);
    }
}
```

1.55 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksSourcesLinkedListMethods.cs

```
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];
    }
}
```

1.56 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksSourcesRecursionlessSizeBalanc

```
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)]
        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);
    }
}
```

1.57 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksSourcesSizeBalancedTreeMetho

```
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;

        [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);
    }
}
```

1.58 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksTargetsRecursionlessSizeBalanc

```
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;

        [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);
    }
}
```

1.59 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksTargetsSizeBalancedTreeMetho

```
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 UInt32InternalLinksTargetsSizeBalancedTreeMethods :
     ↪  UInt32InternalLinksSizeBalancedTreeMethodsBase
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32InternalLinksTargetsSizeBalancedTreeMethods(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;

        [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);
    }
}
```

1.60 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32SplitMemoryLinks.cs

```
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);
                _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)]
        protected override bool LessOrEqualThan(TLink first, TLink second) => first <= second;

        [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 TLink GetZero() => 0U;

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

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

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

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

1.61 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32UnusedLinksListMethods.cs

```
using System.Runtime.CompilerServices;
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 UInt32UnusedLinksListMethods : UnusedLinksListMethods<TLink>
    {
        private readonly RawLinkDataPart<TLink>* _links;
        private readonly LinksHeader<TLink>* _header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32UnusedLinksListMethods(RawLinkDataPart<TLink>* links, LinksHeader<TLink>*
        ↪   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)]
        protected override ref LinksHeader<TLink> GetHeaderReference() => ref *_header;
    }
}
```

1.62 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksRecursionlessSizeBalancedTree

```
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 UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase :
    ↪   ExternalLinksRecursionlessSizeBalancedTreeMethodsBase<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
        ↪   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);
        }

        [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);
            }
        }
    }
```

1.63 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksSizeBalancedTreeMethodsBase

```
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)]
        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);
        }
    }
}
```

1.64 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksSourcesRecursionlessSizeBalan

```
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;

        [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;
        }
    }
}
```

1.65 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksSourcesSizeBalancedTreeMeth

```
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)]
        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;
        }
    }
}
```

1.66 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksTargetsRecursionlessSizeBalanc

```
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 UInt64ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods :
    ↪  UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public
        ↪  UInt64ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(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;

        [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() => Header->RootAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }
    }
}
```

1.67 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksTargetsSizeBalancedTreeMetho

```
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 UInt64ExternalLinksTargetsSizeBalancedTreeMethods :
     ↪  UInt64ExternalLinksSizeBalancedTreeMethodsBase
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64ExternalLinksTargetsSizeBalancedTreeMethods(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;

        [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() => Header->RootAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }
    }
}
```

1.68 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksRecursionlessSizeBalancedTreeM

```
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)]
        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);
    }
}
```

1.69 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksSizeBalancedTreeMethodsBase.

```
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 UInt64InternalLinksSizeBalancedTreeMethodsBase :
     ↪  InternalLinksSizeBalancedTreeMethodsBase<TLink>
    {
        protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
        protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
        protected new readonly LinksHeader<TLink>* Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected UInt64InternalLinksSizeBalancedTreeMethodsBase(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 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);
    }
}
```

1.70 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksSourcesLinkedListMethods.cs

```
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];

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

1.71 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksSourcesRecursionlessSizeBalanc

```
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 UInt64InternalLinksSourcesRecursionlessSizeBalancedTreeMethods :
     ↪  UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public
         ↪  UInt64InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(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);
    }
}
```

1.72 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksSourcesSizeBalancedTreeMetho

```
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 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);
    }
}
```

1.73 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksTargetsRecursionlessSizeBalanc

```
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)]
        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);
    }
}
```

1.74 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksTargetsSizeBalancedTreeMetho

```
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;

        [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);
    }
}
```

1.75 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64SplitMemoryLinks.cs

```
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) { }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64SplitMemoryLinks(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
                 ↪  UInt64InternalLinksSourcesSizeBalancedTreeMethods(Constants,
                 ↪  _linksDataParts, _linksIndexParts, _header);
                _createExternalSourceTreeMethods = () => new
                 ↪  UInt64ExternalLinksSourcesSizeBalancedTreeMethods(Constants,
                 ↪  _linksDataParts, _linksIndexParts, _header);
                _createInternalTargetTreeMethods = () => new
                 ↪  UInt64InternalLinksTargetsSizeBalancedTreeMethods(Constants,
                 ↪  _linksDataParts, _linksIndexParts, _header);
                _createExternalTargetTreeMethods = () => new
                 ↪  UInt64ExternalLinksTargetsSizeBalancedTreeMethods(Constants,
                 ↪  _linksDataParts, _linksIndexParts, _header);
            }
            else
            {
                _createInternalSourceTreeMethods = () => new
                 ↪  UInt64InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(Constants,
                 ↪  _linksDataParts, _linksIndexParts, _header);
                _createExternalSourceTreeMethods = () => new
                 ↪  UInt64ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(Constants,
                 ↪  _linksDataParts, _linksIndexParts, _header);
                _createInternalTargetTreeMethods = () => new
                 ↪  UInt64InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(Constants,
                 ↪  _linksDataParts, _linksIndexParts, _header);
                _createExternalTargetTreeMethods = () => new
                 ↪  UInt64ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(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
                 ↪  UInt64InternalLinksSourcesLinkedListMethods(Constants, _linksDataParts,
                 ↪  _linksIndexParts);
            }
            else
            {
                InternalSourcesTreeMethods = _createInternalSourceTreeMethods();
            }
            ExternalSourcesTreeMethods = _createExternalSourceTreeMethods();
            InternalTargetsTreeMethods = _createInternalTargetTreeMethods();
            ExternalTargetsTreeMethods = _createExternalTargetTreeMethods();
            UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_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(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;
    }
}
```

1.76 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64UnusedLinksListMethods.cs

```
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)]
        protected override ref LinksHeader<TLink> GetHeaderReference() => ref *_header;
    }
}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        /// <summary>
        /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
         ↪  (концом).
        /// </summary>
        /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
        /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
        /// <returns>Индекс искомой связи.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Search(TLink source, TLink target)
        {
            var root = GetTreeRoot();
            while (!EqualToZero(root))
            {
                ref var rootLink = ref GetLinkReference(root);
                var rootSource = rootLink.Source;
                var rootTarget = rootLink.Target;
                if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                 ↪  node.Key < root.Key
                {
                    root = GetLeftOrDefault(root);
                }
                else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
                 ↪  node.Key > root.Key
                {
                    root = GetRightOrDefault(root);
                }
                else // node.Key == root.Key
                {
                    return root;
                }
            }
            return Zero;
        }

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

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

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

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

1.78 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksRecursionlessSizeBalancedTreeMethodsBase

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

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

namespace Platform.Data.Doublets.Memory.United.Generic
{
    public unsafe abstract class LinksRecursionlessSizeBalancedTreeMethodsBase<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* Links;
        protected readonly byte* Header;

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
        {
            ref var link = ref GetLinkReference(linkIndex);
            return new Link<TLink>(linkIndex, link.Source, link.Target);
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
        {
            ref var firstLink = ref GetLinkReference(first);
            ref var secondLink = ref GetLinkReference(second);
            return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
             ↪  secondLink.Source, secondLink.Target);
        }

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

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

        /// <summary>
        /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
         ↪  (концом).
        /// </summary>
        /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
        /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
        /// <returns>Индекс искомой связи.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Search(TLink source, TLink target)
        {
            var root = GetTreeRoot();
            while (!EqualToZero(root))
            {
                ref var rootLink = ref GetLinkReference(root);
                var rootSource = rootLink.Source;
                var rootTarget = rootLink.Target;
                if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                 ↪  node.Key < root.Key
                {
                    root = GetLeftOrDefault(root);
                }
                else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
                 ↪  node.Key > root.Key
                {
                    root = GetRightOrDefault(root);
                }
                else // node.Key == root.Key
                {
```

```
                return root;
            }
        }
        return Zero;
    }

    // TODO: Return indices range instead of references count
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public TLink CountUsages(TLink link)
    {
        var root = GetTreeRoot();
        var total = GetSize(root);
        var totalRightIgnore = Zero;
        while (!EqualToZero(root))
        {
            var @base = GetBasePartValue(root);
            if (LessOrEqualThan(@base, link))
            {
                root = GetRightOrDefault(root);
            }
            else
            {
                totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
                root = GetLeftOrDefault(root);
            }
        }
        root = GetTreeRoot();
        var totalLeftIgnore = Zero;
        while (!EqualToZero(root))
        {
            var @base = GetBasePartValue(root);
            if (GreaterOrEqualThan(@base, link))
            {
                root = GetLeftOrDefault(root);
            }
            else
            {
                totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
                root = GetRightOrDefault(root);
            }
        }
        return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
    }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        /// <summary>
        /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
         ↪  (концом).
        /// </summary>
        /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
        /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
        /// <returns>Индекс искомой связи.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Search(TLink source, TLink target)
        {
            var root = GetTreeRoot();
            while (!EqualToZero(root))
            {
                ref var rootLink = ref GetLinkReference(root);
                var rootSource = rootLink.Source;
                var rootTarget = rootLink.Target;
                if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                 ↪  node.Key < root.Key
                {
                    root = GetLeftOrDefault(root);
                }
                else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
                ↪  node.Key > root.Key
                {
                    root = GetRightOrDefault(root);
                }
                else // node.Key == root.Key
                {
                    return root;
                }
            }
            return Zero;
        }

        // TODO: Return indices range instead of references count
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink CountUsages(TLink link)
        {
            var root = GetTreeRoot();
            var total = GetSize(root);
            var totalRightIgnore = Zero;
            while (!EqualToZero(root))
            {
                var @base = GetBasePartValue(root);
                if (LessOrEqualThan(@base, link))
                {
                    root = GetRightOrDefault(root);
                }
                else
                {
                    totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
                    root = GetLeftOrDefault(root);
                }
            }
            root = GetTreeRoot();
            var totalLeftIgnore = Zero;
            while (!EqualToZero(root))
            {
                var @base = GetBasePartValue(root);
                if (GreaterOrEqualThan(@base, link))
                {
                    root = GetLeftOrDefault(root);
                }
                else
                {
                    totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
                    root = GetRightOrDefault(root);
                }
            }
            return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
        }

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

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

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

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

```
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

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

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

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

        [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;
        }
    }
}
```

1.81 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesRecursionlessSizeBalancedTreeMeth

```
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;

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

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

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

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

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

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

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

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

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

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

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

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

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

```
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.84 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsRecursionlessSizeBalancedTreeMetho

```
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;

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

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

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

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

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

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

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

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

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

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

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

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

```
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        protected readonly IResizableDirectMemory _memory;
        protected readonly long _memoryReservationStep;

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
        {
            var constants = Constants;
            var @break = constants.Break;
            if (restrictions.Count == 0)
            {
                for (var link = GetOne(); LessOrEqualThan(link,
                ↪   GetHeaderReference().AllocatedLinks); link = Increment(link))
                {
                    if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
                    {
                        return @break;
                    }
                }
                return @break;
            }
            var @continue = constants.Continue;
            var any = constants.Any;
            var index = restrictions[constants.IndexPart];
            if (restrictions.Count == 1)
            {
                if (AreEqual(index, any))
                {
                    return Each(handler, Array.Empty<TLink>());
                }
                if (!Exists(index))
                {
                    return @continue;
                }
                return handler(GetLinkStruct(index));
            }
            if (restrictions.Count == 2)
            {
                var value = restrictions[1];
                if (AreEqual(index, any))
                {
                    if (AreEqual(value, any))
                    {
                        return Each(handler, Array.Empty<TLink>());
                    }
                    if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
                    {
                        return @break;
                    }
                    return Each(handler, new Link<TLink>(index, any, value));
                }
                else
                {
                    if (!Exists(index))
                {
                    return @continue;
                }
                if (AreEqual(value, any))
                {
                    return handler(GetLinkStruct(index));
                }
                ref var storedLinkValue = ref GetLinkReference(index);
                if (AreEqual(storedLinkValue.Source, value) ||
                    AreEqual(storedLinkValue.Target, value))
                {
                    return handler(GetLinkStruct(index));
                }
                return @continue;
            }
        }
        if (restrictions.Count == 3)
        {
            var source = restrictions[constants.SourcePart];
            var target = restrictions[constants.TargetPart];
            if (AreEqual(index, any))
            {
                if (AreEqual(source, any) && AreEqual(target, any))
                {
                    return Each(handler, Array.Empty<TLink>());
                }
                else if (AreEqual(source, any))
                {
                    return TargetsTreeMethods.EachUsage(target, handler);
                }
                else if (AreEqual(target, any))
                {
                    return SourcesTreeMethods.EachUsage(source, handler);
                }
                else //if(source != Any && target != Any)
                {
                    var link = SourcesTreeMethods.Search(source, target);
                    return AreEqual(link, constants.Null) ? @continue : handler(GetLinkStruct(link));
                }
            }
            else
            {
                if (!Exists(index))
                {
                    return @continue;
                }
                if (AreEqual(source, any) && AreEqual(target, any))
                {
                    return handler(GetLinkStruct(index));
                }
                ref var storedLinkValue = ref GetLinkReference(index);
                if (!AreEqual(source, any) && !AreEqual(target, any))
                {
                    if (AreEqual(storedLinkValue.Source, source) &&
                        AreEqual(storedLinkValue.Target, target))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    return @continue;
                }
                var value = default(TLink);
                if (AreEqual(source, any))
                {
                    value = target;
                }
                if (AreEqual(target, any))
                {
                    value = source;
                }
                if (AreEqual(storedLinkValue.Source, value) ||
                    AreEqual(storedLinkValue.Target, value))
                {
                    return handler(GetLinkStruct(index));
                }
                return @continue;
            }
        }
```

```
                throw new NotSupportedException("Другие размеры и способы ограничений не
                ↪  поддерживаются.");
            }

            /// <remarks>
            /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
            ↪  в другом месте (но не в менеджере памяти, а в логике Links)
            /// </remarks>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
            {
                var constants = Constants;
                var @null = constants.Null;
                var linkIndex = restrictions[constants.IndexPart];
                ref var link = ref GetLinkReference(linkIndex);
                ref var header = ref GetHeaderReference();
                ref var firstAsSource = ref header.RootAsSource;
                ref var firstAsTarget = ref header.RootAsTarget;
                // Будет корректно работать только в том случае, если пространство выделенной связи
                ↪  предварительно заполнено нулями
                if (!AreEqual(link.Source, @null))
                {
                    SourcesTreeMethods.Detach(ref firstAsSource, linkIndex);
                }
                if (!AreEqual(link.Target, @null))
                {
                    TargetsTreeMethods.Detach(ref firstAsTarget, linkIndex);
                }
                link.Source = substitution[constants.SourcePart];
                link.Target = substitution[constants.TargetPart];
                if (!AreEqual(link.Source, @null))
                {
                    SourcesTreeMethods.Attach(ref firstAsSource, linkIndex);
                }
                if (!AreEqual(link.Target, @null))
                {
                    TargetsTreeMethods.Attach(ref firstAsTarget, linkIndex);
                }
                return linkIndex;
            }

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

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

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

            /// <remarks>
            /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
            ↪  адрес реально поменялся
            ///
            /// Указатель this.links может быть в том же месте,
            /// так как 0-я связь не используется и имеет такой же размер как Header,
            /// поэтому header размещается в том же месте, что и 0-я связь
            /// </remarks>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected abstract void SetPointers(IResizableDirectMemory memory);

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

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

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

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

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

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        protected virtual TLink GetZero() => default;

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

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

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

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

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

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

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

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

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

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

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

        #region Disposable

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

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

        #endregion
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Memory.United.Generic
{
    public unsafe class UnusedLinksListMethods<TLink> :
     ↪  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;
    }
}
```

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

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

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

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

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

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

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

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

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

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

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

1.90 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksRecursionlessSizeBalancedTreeMetho

```
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;

        [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];
    }
}
```

1.91 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksSizeBalancedTreeMethodsBase.cs

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

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

namespace Platform.Data.Doublets.Memory.United.Specific
{
    public unsafe abstract class 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)]
        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];
    }
}
```

1.92 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksSourcesRecursionlessSizeBalancedTre

```
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 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;
        }
    }
}
```

1.93 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksSourcesSizeBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Memory.United.Specific
{
    public unsafe class 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;

        [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;
        }
    }
}
```

1.94 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksTargetsRecursionlessSizeBalancedTre

```
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 UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods :
    ↪  UInt32LinksRecursionlessSizeBalancedTreeMethodsBase
    {
        public UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods(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)]
        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;
        }
    }
}
```

1.95 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksTargetsSizeBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Memory.United.Specific
{
    public unsafe class 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)]
        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;
        }
    }
}
```

1.96 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32UnitedMemoryLinks.cs

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

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

namespace Platform.Data.Doublets.Memory.United.Specific
{
    /// <summary>
    /// <para>Represents a low-level implementation of direct access to resizable memory, for
     ↪  organizing the storage of links with addresses represented as <see cref="uint" />.</para>
    /// <para>Представляет низкоуровневую реализация прямого доступа к памяти с переменным
     ↪  размером, для организации хранения связей с адресами представленными в виде <see
     ↪  cref="uint"/>.</para>
    /// </summary>
    public unsafe class UInt32UnitedMemoryLinks : UnitedMemoryLinksBase<uint>
    {
        private readonly Func<ILinksTreeMethods<uint>> _createSourceTreeMethods;
        private readonly Func<ILinksTreeMethods<uint>> _createTargetTreeMethods;
        private LinksHeader<uint>* _header;
        private RawLink<uint>* _links;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32UnitedMemoryLinks(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 UInt32UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
         ↪  FileMappedResizableDirectMemory(address, memoryReservationStep),
         ↪  memoryReservationStep) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
         ↪  DefaultLinksSizeStep) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32UnitedMemoryLinks(IResizableDirectMemory memory, long
         ↪  memoryReservationStep) : this(memory, memoryReservationStep,
         ↪  Default<LinksConstants<uint>>.Instance, IndexTreeType.Default) { }

        [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)]
        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;
    }
}
```

1.97 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32UnusedLinksListMethods.cs

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

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

namespace Platform.Data.Doublets.Memory.United.Specific
{
    public unsafe class 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;
    }
}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
    }
}
```

1.99 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksRecursionlessSizeBalancedTreeMetho

```
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)]
        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];
    }
}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.102 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesRecursionlessSizeBalancedTr

```
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 UInt64LinksSourcesRecursionlessSizeBalancedTreeMethods :
     ↪  UInt64LinksRecursionlessSizeBalancedTreeMethodsBase
    {
        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;
        }
    }
}
```

1.103 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesSizeBalancedTreeMethods.c

```
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.105 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsRecursionlessSizeBalancedTr

```
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 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;
        }
    }
}
```

1.106 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsSizeBalancedTreeMethods.c

```
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        [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;
    }
}
```

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

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

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

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

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

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

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

1.109 ./csharp/Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs

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

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

namespace Platform.Data.Doublets.PropertyOperators
{
    public class PropertiesOperator<TLink> : LinksOperatorBase<TLink>, IProperties<TLink, TLink,
     ↪  TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
         ↪  EqualityComparer<TLink>.Default;
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public PropertiesOperator(ILinks<TLink> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink GetValue(TLink @object, TLink property)
        {
            var links = _links;
            var objectProperty = links.SearchOrDefault(@object, property);
            if (_equalityComparer.Equals(objectProperty, default))
            {
                return default;
            }
            var constants = links.Constants;
            var any = constants.Any;
            var query = new Link<TLink>(any, objectProperty, any);
            var valueLink = links.SingleOrDefault(query);
            if (valueLink == null)
            {
                return default;
            }
            return links.GetTarget(valueLink[constants.IndexPart]);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void SetValue(TLink @object, TLink property, TLink value)
        {
            var links = _links;
            var objectProperty = links.GetOrCreate(@object, property);
            links.DeleteMany(links.AllIndices(links.Constants.Any, objectProperty));
            links.GetOrCreate(objectProperty, value);
        }
    }
}
```

1.110 ./csharp/Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs

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

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

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

        private readonly TLink _propertyMarker;
        private readonly TLink _propertyValueMarker;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public PropertyOperator(ILinks<TLink> links, TLink propertyMarker, TLink
         ↪  propertyValueMarker) : base(links)
        {
            _propertyMarker = propertyMarker;
            _propertyValueMarker = propertyValueMarker;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Get(TLink link)
        {
            var property = _links.SearchOrDefault(link, _propertyMarker);
            return GetValue(GetContainer(property));
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private TLink GetContainer(TLink property)
        {
            var valueContainer = default(TLink);
            if (_equalityComparer.Equals(property, default))
            {
                return valueContainer;
            }
            var links = _links;
            var constants = links.Constants;
            var countinueConstant = constants.Continue;
            var breakConstant = constants.Break;
            var anyConstant = constants.Any;
```

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

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

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

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

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

        private readonly TLink _stack;

        public bool IsEmpty
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _equalityComparer.Equals(Peek(), _stack);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Stack(ILinks<TLink> links, TLink stack) : base(links) => _stack = stack;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private TLink GetStackMarker() => _links.GetSource(_stack);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private TLink GetTop() => _links.GetTarget(_stack);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Peek() => _links.GetTarget(GetTop());

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Pop()
        {
            var element = Peek();
            if (!_equalityComparer.Equals(element, _stack))
            {
                var top = GetTop();
                var previousTop = _links.GetSource(top);
                _links.Update(_stack, GetStackMarker(), previousTop);
```

```
                    _links.Delete(top);
                }
                return element;
            }

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

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

```
using System.Runtime.CompilerServices;

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

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

1.113 ./csharp/Platform.Data.Doublets/SynchronizedLinks.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Data.Doublets;
using Platform.Threading.Synchronization;

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

namespace Platform.Data.Doublets
{
    /// <remarks>
    /// TODO: Autogeneration of synchronized wrapper (decorator).
    /// TODO: Try to unfold code of each method using IL generation for performance improvements.
    /// TODO: Or even to unfold multiple layers of implementations.
    /// </remarks>
    public class SynchronizedLinks<TLinkAddress> : ISynchronizedLinks<TLinkAddress>
    {
        public LinksConstants<TLinkAddress> Constants
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        public ISynchronization SyncRoot
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        public ILinks<TLinkAddress> Sync
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        public ILinks<TLinkAddress> Unsync
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SynchronizedLinks(ILinks<TLinkAddress> links) : this(new
        ↪   ReaderWriterLockSynchronization(), links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SynchronizedLinks(ISynchronization synchronization, ILinks<TLinkAddress> links)
        {
            SyncRoot = synchronization;
            Sync = this;
            Unsync = links;
            Constants = links.Constants;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Count(IList<TLinkAddress> restriction) =>
         ↪  SyncRoot.ExecuteReadOperation(restriction, Unsync.Count);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler,
         ↪  IList<TLinkAddress> restrictions) => SyncRoot.ExecuteReadOperation(handler,
         ↪  restrictions, (handler1, restrictions1) => Unsync.Each(handler1, restrictions1));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Create(IList<TLinkAddress> restrictions) =>
         ↪  SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Create);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress>
         ↪  substitution) => SyncRoot.ExecuteWriteOperation(restrictions, substitution,
         ↪  Unsync.Update);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Delete(IList<TLinkAddress> restrictions) =>
         ↪  SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Delete);

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

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

1.114 ./csharp/Platform.Data.Doublets/UInt64LinksExtensions.cs

```
using System;
using System.Text;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Singletons;

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

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

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

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

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

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

1.115 ./csharp/Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs

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

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

namespace Platform.Data.Doublets
{
    public class UInt64LinksTransactionsLayer : LinksDisposableDecoratorBase<ulong> //-V3073
    {
        /// <remarks>
        /// Альтернативные варианты хранения трансформации (элемента транзакции):
        ///
        /// private enum TransitionType
        /// {
        ///     Creation,
        ///     UpdateOf,
        ///     UpdateTo,
        ///     Deletion
        /// }
        ///
        /// private struct Transition
        /// {
        ///     public ulong TransactionId;
        ///     public UniqueTimestamp Timestamp;
        ///     public TransactionItemType Type;
        ///     public Link Source;
        ///     public Link Linker;
        ///     public Link Target;
        /// }
        ///
        /// Или
        ///
        /// public struct TransitionHeader
        /// {
        ///     public ulong TransactionIdCombined;
        ///     public ulong TimestampCombined;
        ///
        ///     public ulong TransactionId
        ///     {
        ///         get
        ///         {
        ///             return (ulong) mask &amp; TransactionIdCombined;
        ///         }
        ///     }
        ///
        ///     public UniqueTimestamp Timestamp
        ///     {
        ///         get
        ///         {
        ///             return (UniqueTimestamp)mask &amp; TransactionIdCombined;
        ///         }
        ///     }
        ///
        ///     public TransactionItemType Type
        ///     {
        ///         get
        ///         {
        ///             // Использовать по одному биту из TransactionId и Timestamp,
        ///             // для значения в 2 бита, которое представляет тип операции
        ///             throw new NotImplementedException();
        ///         }
        ///     }
        /// }
        ///
        /// private struct Transition
        /// {
        ///     public TransitionHeader Header;
        ///     public Link Source;
        ///     public Link Linker;
        ///     public Link Target;
        /// }
        ///
        /// </remarks>
        public struct Transition : IEquatable<Transition>
        {
            public static readonly long Size = Structure<Transition>.Size;

            public readonly ulong TransactionId;
            public readonly Link<ulong> Before;
            public readonly Link<ulong> After;
            public readonly Timestamp Timestamp;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
             ↪  transactionId, Link<ulong> before, Link<ulong> after)
            {
                TransactionId = transactionId;
                Before = before;
                After = after;
                Timestamp = uniqueTimestampFactory.Create();
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
             ↪  transactionId, Link<ulong> before) : this(uniqueTimestampFactory, transactionId,
             ↪  before, default) { }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
             ↪  transactionId) : this(uniqueTimestampFactory, transactionId, default, default) {
             ↪  }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
             ↪  {After}";

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public override bool Equals(object obj) => obj is Transition transition ?
             ↪  Equals(transition) : false;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public override int GetHashCode() => (TransactionId, Before, After,
             ↪  Timestamp).GetHashCode();

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public bool Equals(Transition other) => TransactionId == other.TransactionId &&
             ↪  Before == other.Before && After == other.After && Timestamp == other.Timestamp;

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

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

        /// <remarks>
        /// Другие варианты реализации транзакций (атомарности):
        ///     1. Разделение хранения значения связи ((Source Target) или (Source Linker
        ↪  Target)) и индексов.
        ///     2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
        ↪  потребуется решить вопрос
        ///        со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
        ↪  пересечениями идентификаторов.
        ///
        /// Где хранить промежуточный список транзакций?
        ///
        /// В оперативной памяти:
        ///  Минусы:
        ///     1. Может усложнить систему, если она будет функционировать самостоятельно,
        ///     так как нужно отдельно выделять память под список трансформаций.
        ///     2. Выделенной оперативной памяти может не хватить, в том случае,
        ///     если транзакция использует слишком много трансформаций.
        ///         -> Можно использовать жёсткий диск для слишком длинных транзакций.
        ///         -> Максимальный размер списка трансформаций можно ограничить / задать
        ↪  константой.
        ///     3. При подтверждении транзакции (Commit) все трансформации записываются разом
        ↪  создавая задержку.
        ///
        /// На жёстком диске:
        ///  Минусы:
        ///     1. Длительный отклик, на запись каждой трансформации.
        ///     2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
        ///         -> Это может решаться упаковкой/исключением дублирующих операций.
        ///         -> Также это может решаться тем, что короткие транзакции вообще
        ///            не будут записываться в случае отката.
        ///     3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
        ↪  операции (трансформации)
        ///        будут записаны в лог.
        ///
        /// </remarks>
        public class Transaction : DisposableBase
        {
            private readonly Queue<Transition> _transitions;
            private readonly UInt64LinksTransactionsLayer _layer;
            public bool IsCommitted { get; private set; }
            public bool IsReverted { get; private set; }

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

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

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

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
            {
                if (transaction.IsReverted)
                {
                    throw new InvalidOperationException("Transation is reverted.");
                }
                if (transaction.IsCommitted)
                {
                    throw new InvalidOperationException("Transation is commited.");
                }
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override void Dispose(bool manual, bool wasDisposed)
            {
                if (!wasDisposed && _layer != null && !_layer.Disposable.IsDisposed)
                {
                    if (!IsCommitted && !IsReverted)
                    {
                        Revert();
                    }
                    _layer.ResetCurrentTransation();
                }
            }
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
            : base(links)
        {
            if (string.IsNullOrWhiteSpace(logAddress))
            {
                throw new ArgumentNullException(nameof(logAddress));
            }
            // В первой строке файла хранится последняя закоммиченную транзакцию.
            // При запуске это используется для проверки удачного закрытия файла лога.
            // In the first line of the file the last committed transaction is stored.
            // On startup, this is used to check that the log file is successfully closed.
            var lastCommitedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
            var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
            if (!lastCommitedTransition.Equals(lastWrittenTransition))
            {
                Dispose();
                throw new NotSupportedException("Database is damaged, autorecovery is not
                ↪  supported yet.");
            }
            if (lastCommitedTransition == default)
            {
                FileHelpers.WriteFirst(logAddress, lastCommitedTransition);
            }
            _lastCommitedTransition = lastCommitedTransition;
            // TODO: Think about a better way to calculate or store this value
            var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
            _lastCommitedTransactionId = allTransitions.Length > 0 ? allTransitions.Max(x =>
            ↪  x.TransactionId) : 0;
            _uniqueTimestampFactory = new UniqueTimestampFactory();
            _logAddress = logAddress;
            _log = FileHelpers.Append(logAddress);
            _transitions = new Queue<Transition>();
            _transitionsPusher = new Task(TransitionsPusher);
            _transitionsPusher.Start();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IList<ulong> GetLinkValue(ulong link) => _links.GetLink(link);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override ulong Create(IList<ulong> restrictions)
        {
            var createdLinkIndex = _links.Create();
            var createdLink = new Link<ulong>(_links.GetLink(createdLinkIndex));
            CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
            ↪  default, createdLink));
            return createdLinkIndex;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
        {
            var linkIndex = restrictions[_constants.IndexPart];
            var beforeLink = new Link<ulong>(_links.GetLink(linkIndex));
            linkIndex = _links.Update(restrictions, substitution);
            var afterLink = new Link<ulong>(_links.GetLink(linkIndex));
            CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
            ↪  beforeLink, afterLink));
            return linkIndex;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override void Delete(IList<ulong> restrictions)
        {
            var link = restrictions[_constants.IndexPart];
            var deletedLink = new Link<ulong>(_links.GetLink(link));
            _links.Delete(link);
            CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
            ↪  deletedLink, default));
        }

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void ResetCurrentTransation()
        {
            _currentTransactionId = 0;
            _currentTransactionTransitions = null;
            _currentTransaction = null;
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void TransitionsPusher()
        {
            while (!Disposable.IsDisposed && _transitionsPusher != null)
            {
                Thread.Sleep(DefaultPushDelay);
                PushTransitions();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Transaction BeginTransaction() => new Transaction(this);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void DisposeTransitions()
        {
            try
            {
                var pusher = _transitionsPusher;
                if (pusher != null)
                {
                    _transitionsPusher = null;
                    pusher.Wait();
                }
                if (_transitions != null)
                {
                    PushTransitions();
                }
                _log.DisposeIfPossible();
                FileHelpers.WriteFirst(_logAddress, _lastCommitedTransition);
            }
            catch (Exception ex)
            {
                ex.Ignore();
            }
        }

        #region DisposalBase

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

        #endregion
    }
}
```

1.116 ./csharp/Platform.Data.Doublets.Tests/GenericLinksTests.cs

```
using System;
using Xunit;
using Platform.Reflection;
using Platform.Memory;
using Platform.Scopes;
using Platform.Data.Doublets.Memory.United.Generic;

namespace Platform.Data.Doublets.Tests
{
    public unsafe static class GenericLinksTests
    {
        [Fact]
        public static void CRUDTest()
        {
            Using<byte>(links => links.TestCRUDOperations());
            Using<ushort>(links => links.TestCRUDOperations());
            Using<uint>(links => links.TestCRUDOperations());
            Using<ulong>(links => links.TestCRUDOperations());
        }

        [Fact]
        public static void RawNumbersCRUDTest()
        {
            Using<byte>(links => links.TestRawNumbersCRUDOperations());
            Using<ushort>(links => links.TestRawNumbersCRUDOperations());
            Using<uint>(links => links.TestRawNumbersCRUDOperations());
            Using<ulong>(links => links.TestRawNumbersCRUDOperations());
        }

        [Fact]
        public static void MultipleRandomCreationsAndDeletionsTest()
        {
            Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
            ↪  MultipleRandomCreationsAndDeletions(16)); // Cannot use more because current
            ↪  implementation of tree cuts out 5 bits from the address space.
            Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Te
            ↪  stMultipleRandomCreationsAndDeletions(100));
            Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
            ↪  MultipleRandomCreationsAndDeletions(100));
            Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
            ↪  tMultipleRandomCreationsAndDeletions(100));
        }

        private static void Using<TLink>(Action<ILinks<TLink>> action)
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪  UnitedMemoryLinks<TLink>>>())
            {
                action(scope.Use<ILinks<TLink>>());
            }
        }
    }
}
```

1.117 ./csharp/Platform.Data.Doublets.Tests/LinksConstantsTests.cs

```
using Xunit;

namespace Platform.Data.Doublets.Tests
{
    public static class LinksConstantsTests
    {
        [Fact]
        public static void ExternalReferencesTest()
        {
            LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
            ↪  (long.MaxValue + 1UL, ulong.MaxValue));

            //var minimum = new Hybrid<ulong>(0, isExternal: true);
            var minimum = new Hybrid<ulong>(1, isExternal: true);
            var maximum = new Hybrid<ulong>(long.MaxValue, isExternal: true);

            Assert.True(constants.IsExternalReference(minimum));
            Assert.True(constants.IsExternalReference(maximum));
        }
    }
}
```

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

```
using System.IO;
using Xunit;
using Platform.Singletons;
using Platform.Memory;
using Platform.Data.Doublets.Memory.United.Specific;

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

        [Fact]
        public static void BasicFileMappedMemoryTest()
        {
            var tempFilename = Path.GetTempFileName();
            using (var memoryAdapter = new UInt64UnitedMemoryLinks(tempFilename))
            {
                memoryAdapter.TestBasicMemoryOperations();
            }
            File.Delete(tempFilename);
        }

        [Fact]
        public static void BasicHeapMemoryTest()
        {
            using (var memory = new
             ↪  HeapResizableDirectMemory(UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
            using (var memoryAdapter = new UInt64UnitedMemoryLinks(memory,
             ↪  UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
            {
                memoryAdapter.TestBasicMemoryOperations();
            }
        }

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

        [Fact]
        public static void NonexistentReferencesHeapMemoryTest()
        {
            using (var memory = new
             ↪  HeapResizableDirectMemory(UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
            using (var memoryAdapter = new UInt64UnitedMemoryLinks(memory,
             ↪  UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
            {
                memoryAdapter.TestNonexistentReferences();
            }
        }

        private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
        {
            var link = memoryAdapter.Create();
            memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
            var resultLink = _constants.Null;
            memoryAdapter.Each(foundLink =>
            {
                resultLink = foundLink[_constants.IndexPart];
                return _constants.Break;
            }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
            Assert.True(resultLink == link);
            Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
            memoryAdapter.Delete(link);
        }
    }
}
```

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

```
using Xunit;
using Platform.Scopes;
using Platform.Memory;
using Platform.Data.Doublets.Decorators;
using Platform.Reflection;
using Platform.Data.Doublets.Memory.United.Generic;
using Platform.Data.Doublets.Memory.United.Specific;

namespace Platform.Data.Doublets.Tests
{
    public static class ScopeTests
    {
        [Fact]
        public static void SingleDependencyTest()
        {
            using (var scope = new Scope())
            {
                scope.IncludeAssemblyOf<IMemory>();
                var instance = scope.Use<IDirectMemory>();
                Assert.IsType<HeapResizableDirectMemory>(instance);
            }
        }

        [Fact]
        public static void CascadeDependencyTest()
        {
            using (var scope = new Scope())
            {
                scope.Include<TemporaryFileMappedResizableDirectMemory>();
                scope.Include<UInt64UnitedMemoryLinks>();
                var instance = scope.Use<ILinks<ulong>>();
                Assert.IsType<UInt64UnitedMemoryLinks>(instance);
            }
        }

        [Fact(Skip = "Would be fixed later.")]
        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);
            }
        }
    }
}
```

1.120 ./csharp/Platform.Data.Doublets.Tests/SplitMemoryGenericLinksTests.cs

```
using System;
using Xunit;
using Platform.Memory;
using Platform.Data.Doublets.Memory.Split.Generic;
using Platform.Data.Doublets.Memory;

namespace Platform.Data.Doublets.Tests
{
    public unsafe static class SplitMemoryGenericLinksTests
    {
        [Fact]
        public static void CRUDTest()
        {
            Using<byte>(links => links.TestCRUDOperations());
            Using<ushort>(links => links.TestCRUDOperations());
            Using<uint>(links => links.TestCRUDOperations());
            Using<ulong>(links => links.TestCRUDOperations());
        }

        [Fact]
        public static void RawNumbersCRUDTest()
        {
            UsingWithExternalReferences<byte>(links => links.TestRawNumbersCRUDOperations());
            UsingWithExternalReferences<ushort>(links => links.TestRawNumbersCRUDOperations());
            UsingWithExternalReferences<uint>(links => links.TestRawNumbersCRUDOperations());
            UsingWithExternalReferences<ulong>(links => links.TestRawNumbersCRUDOperations());
        }

        [Fact]
        public static void MultipleRandomCreationsAndDeletionsTest()
        {
            Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
                MultipleRandomCreationsAndDeletions(16)); // Cannot use more because current
                implementation of tree cuts out 5 bits from the address space.
            Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Te
                stMultipleRandomCreationsAndDeletions(100));
            Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
                MultipleRandomCreationsAndDeletions(100));
            Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
                tMultipleRandomCreationsAndDeletions(100));
        }

        private static void Using<TLink>(Action<ILinks<TLink>> action)
        {
            using (var dataMemory = new HeapResizableDirectMemory())
            using (var indexMemory = new HeapResizableDirectMemory())
            using (var memory = new SplitMemoryLinks<TLink>(dataMemory, indexMemory))
            {
                action(memory);
            }
        }

        private static void UsingWithExternalReferences<TLink>(Action<ILinks<TLink>> action)
        {
            var contants = new LinksConstants<TLink>(enableExternalReferencesSupport: true);
            using (var dataMemory = new HeapResizableDirectMemory())
            using (var indexMemory = new HeapResizableDirectMemory())
            using (var memory = new SplitMemoryLinks<TLink>(dataMemory, indexMemory,
                SplitMemoryLinks<TLink>.DefaultLinksSizeStep, contants))
            {
                action(memory);
            }
        }
    }
}
```

1.121 ./csharp/Platform.Data.Doublets.Tests/SplitMemoryUInt32LinksTests.cs

```
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());
        }

        [Fact]
        public static void MultipleRandomCreationsAndDeletionsTest()
        {
            Using(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultip
                leRandomCreationsAndDeletions(500));
        }

        private static void Using(Action<ILinks<TLink>> action)
        {
            using (var dataMemory = new HeapResizableDirectMemory())
            using (var indexMemory = new HeapResizableDirectMemory())
            using (var memory = new UInt32SplitMemoryLinks(dataMemory, indexMemory))
            {
                action(memory);
            }
        }

        private static void UsingWithExternalReferences(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 UInt32SplitMemoryLinks(dataMemory, indexMemory,
             ↪  UInt32SplitMemoryLinks.DefaultLinksSizeStep, contants))
            {
                action(memory);
            }
        }
    }
}
```

1.122 ./csharp/Platform.Data.Doublets.Tests/SplitMemoryUInt64LinksTests.cs

```
using System;
using Xunit;
using Platform.Memory;
using Platform.Data.Doublets.Memory.Split.Specific;
using TLink = System.UInt64;

namespace Platform.Data.Doublets.Tests
{
    public unsafe static class SplitMemoryUInt64LinksTests
    {
        [Fact]
        public static void CRUDTest()
        {
            Using(links => links.TestCRUDOperations());
        }

        [Fact]
        public static void RawNumbersCRUDTest()
        {
            UsingWithExternalReferences(links => links.TestRawNumbersCRUDOperations());
        }

        [Fact]
        public static void MultipleRandomCreationsAndDeletionsTest()
        {
            Using(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultip⌋
             ↪  leRandomCreationsAndDeletions(500));
        }

        private static void Using(Action<ILinks<TLink>> action)
        {
            using (var dataMemory = new HeapResizableDirectMemory())
            using (var indexMemory = new HeapResizableDirectMemory())
            using (var memory = new UInt64SplitMemoryLinks(dataMemory, indexMemory))
            {
                action(memory);
            }
        }

        private static void UsingWithExternalReferences(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 UInt64SplitMemoryLinks(dataMemory, indexMemory,
             ↪  UInt64SplitMemoryLinks.DefaultLinksSizeStep, contants))
            {
                action(memory);
            }
        }
    }
}
```

1.123 ./csharp/Platform.Data.Doublets.Tests/TestExtensions.cs

```csharp
using System.Collections.Generic;
using Xunit;
using Platform.Ranges;
using Platform.Numbers;
using Platform.Random;
using Platform.Setters;
using Platform.Converters;

namespace Platform.Data.Doublets.Tests
{
    public static class TestExtensions
    {
        public static void TestCRUDOperations<T>(this ILinks<T> links)
        {
            var constants = links.Constants;

            var equalityComparer = EqualityComparer<T>.Default;

            var zero = default(T);
            var one = Arithmetic.Increment(zero);

            // Create Link
            Assert.True(equalityComparer.Equals(links.Count(), zero));

            var setter = new Setter<T>(constants.Null);
            links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);

            Assert.True(equalityComparer.Equals(setter.Result, constants.Null));

            var linkAddress = links.Create();

            var link = new Link<T>(links.GetLink(linkAddress));

            Assert.True(link.Count == 3);
            Assert.True(equalityComparer.Equals(link.Index, linkAddress));
            Assert.True(equalityComparer.Equals(link.Source, constants.Null));
            Assert.True(equalityComparer.Equals(link.Target, constants.Null));

            Assert.True(equalityComparer.Equals(links.Count(), one));

            // Get first link
            setter = new Setter<T>(constants.Null);
            links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);

            Assert.True(equalityComparer.Equals(setter.Result, linkAddress));

            // Update link to reference itself
            links.Update(linkAddress, linkAddress, linkAddress);

            link = new Link<T>(links.GetLink(linkAddress));

            Assert.True(equalityComparer.Equals(link.Source, linkAddress));
            Assert.True(equalityComparer.Equals(link.Target, linkAddress));

            // Update link to reference null (prepare for delete)
            var updated = links.Update(linkAddress, constants.Null, constants.Null);

            Assert.True(equalityComparer.Equals(updated, linkAddress));

            link = new Link<T>(links.GetLink(linkAddress));

            Assert.True(equalityComparer.Equals(link.Source, constants.Null));
            Assert.True(equalityComparer.Equals(link.Target, constants.Null));

            // Delete link
            links.Delete(linkAddress);

            Assert.True(equalityComparer.Equals(links.Count(), zero));

            setter = new Setter<T>(constants.Null);
            links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);

            Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
        }

        public static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
        {
            // Constants
            var constants = links.Constants;
            var equalityComparer = EqualityComparer<T>.Default;

            var zero = default(T);
            var one = Arithmetic.Increment(zero);
            var two = Arithmetic.Increment(one);

            var h106E = new Hybrid<T>(106L, isExternal: true);
            var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
            var h108E = new Hybrid<T>(-108L);

            Assert.Equal(106L, h106E.AbsoluteValue);
            Assert.Equal(107L, h107E.AbsoluteValue);
            Assert.Equal(108L, h108E.AbsoluteValue);

            // Create Link (External -> External)
            var linkAddress1 = links.Create();

            links.Update(linkAddress1, h106E, h108E);

            var link1 = new Link<T>(links.GetLink(linkAddress1));

            Assert.True(equalityComparer.Equals(link1.Source, h106E));
            Assert.True(equalityComparer.Equals(link1.Target, h108E));

            // Create Link (Internal -> External)
            var linkAddress2 = links.Create();

            links.Update(linkAddress2, linkAddress1, h108E);

            var link2 = new Link<T>(links.GetLink(linkAddress2));

            Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
            Assert.True(equalityComparer.Equals(link2.Target, h108E));

            // Create Link (Internal -> Internal)
            var linkAddress3 = links.Create();

            links.Update(linkAddress3, linkAddress1, linkAddress2);

            var link3 = new Link<T>(links.GetLink(linkAddress3));

            Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
            Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));

            // Search for created link
            var setter1 = new Setter<T>(constants.Null);
            links.Each(h106E, h108E, setter1.SetAndReturnFalse);

            Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));

            // Search for nonexistent link
            var setter2 = new Setter<T>(constants.Null);
            links.Each(h106E, h107E, setter2.SetAndReturnFalse);

            Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));

            // Update link to reference null (prepare for delete)
            var updated = links.Update(linkAddress3, constants.Null, constants.Null);

            Assert.True(equalityComparer.Equals(updated, linkAddress3));

            link3 = new Link<T>(links.GetLink(linkAddress3));

            Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
            Assert.True(equalityComparer.Equals(link3.Target, constants.Null));

            // Delete link
            links.Delete(linkAddress3);

            Assert.True(equalityComparer.Equals(links.Count(), two));

            var setter3 = new Setter<T>(constants.Null);
            links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);

            Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
        }

        public static void TestMultipleRandomCreationsAndDeletions<TLink>(this ILinks<TLink>
         ↪  links, int maximumOperationsPerCycle)
        {
            var comparer = Comparer<TLink>.Default;
            var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
            var uInt64ToAddressConverter = CheckedConverter<ulong, TLink>.Default;
            for (var N = 1; N < maximumOperationsPerCycle; N++)
            {
                var random = new System.Random(N);
                var created = 0UL;
                var deleted = 0UL;
                for (var i = 0; i < N; i++)
                {
                    var linksCount = addressToUInt64Converter.Convert(links.Count());
                    var createPoint = random.NextBoolean();
                    if (linksCount >= 2 && createPoint)
                    {
                        var linksAddressRange = new Range<ulong>(1, linksCount);
                        TLink source = uInt64ToAddressConverter.Convert(random.NextUInt64(linksA
                        ↪   ddressRange));
                        TLink target = uInt64ToAddressConverter.Convert(random.NextUInt64(linksA
                        ↪   ddressRange));
                        ↪   //-V3086
                        var resultLink = links.GetOrCreate(source, target);
                        if (comparer.Compare(resultLink,
                        ↪   uInt64ToAddressConverter.Convert(linksCount)) > 0)
                        {
                            created++;
                        }
                    }
                    else
                    {
                        links.Create();
                        created++;
                    }
                }
                Assert.True(created == addressToUInt64Converter.Convert(links.Count()));
                for (var i = 0; i < N; i++)
                {
                    TLink link = uInt64ToAddressConverter.Convert((ulong)i + 1UL);
                    if (links.Exists(link))
                    {
                        links.Delete(link);
                        deleted++;
                    }
                }
                Assert.True(addressToUInt64Converter.Convert(links.Count()) == 0L);
            }
        }
    }
}
```

1.124 ./csharp/Platform.Data.Doublets.Tests/Uint64LinksExtensionsTests.cs

```
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);
        }
    }
}
```

1.125 ./csharp/Platform.Data.Doublets.Tests/UnitedMemoryUInt32LinksTests.cs

```
using System;
using Xunit;
using Platform.Reflection;
using Platform.Memory;
using Platform.Scopes;
using Platform.Data.Doublets.Memory.United.Specific;
using TLink = System.UInt32;

namespace Platform.Data.Doublets.Tests
{
    public unsafe static class UnitedMemoryUInt32LinksTests
    {
        [Fact]
        public static void CRUDTest()
        {
            Using(links => links.TestCRUDOperations());
        }

        [Fact]
        public static void RawNumbersCRUDTest()
        {
            Using(links => links.TestRawNumbersCRUDOperations());
        }

        [Fact]
        public static void MultipleRandomCreationsAndDeletionsTest()
        {
            Using(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultip
             ↪  leRandomCreationsAndDeletions(100));
        }

        private static void Using(Action<ILinks<TLink>> action)
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
             ↪  UInt32UnitedMemoryLinks>>())
            {
                action(scope.Use<ILinks<TLink>>());
            }
        }
    }
}
```

1.126 ./csharp/Platform.Data.Doublets.Tests/UnitedMemoryUInt64LinksTests.cs

```
using System;
using Xunit;
using Platform.Reflection;
using Platform.Memory;
using Platform.Scopes;
using Platform.Data.Doublets.Memory.United.Specific;
using TLink = System.UInt64;

namespace Platform.Data.Doublets.Tests
{
    public unsafe static class UnitedMemoryUInt64LinksTests
    {
        [Fact]
        public static void CRUDTest()
        {
            Using(links => links.TestCRUDOperations());
        }

        [Fact]
        public static void RawNumbersCRUDTest()
        {
            Using(links => links.TestRawNumbersCRUDOperations());
        }

        [Fact]
        public static void MultipleRandomCreationsAndDeletionsTest()
        {
            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