1.1 ./csharp/Platform.Data.Doublets/CriterionMatchers/TargetMatcher.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.CriterionMatchers
{
    /// <summary>
    /// <para>
    /// Represents the target matcher.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
    /// <seealso cref="ICriterionMatcher{TLinkAddress}"/>
    public class TargetMatcher<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
    ↪  ICriterionMatcher<TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;
        private readonly TLinkAddress _targetToMatch;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="TargetMatcher"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="targetToMatch">
        /// <para>A target to match.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TargetMatcher(ILinks<TLinkAddress> links, TLinkAddress targetToMatch) :
        ↪  base(links) => _targetToMatch = targetToMatch;

        /// <summary>
        /// <para>
        /// Determines whether this instance is matched.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool IsMatched(TLinkAddress link) =>
        ↪  _equalityComparer.Equals(_links.GetTarget(link), _targetToMatch);
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Decorators
{
    /// <summary>
    /// <para>
    /// Represents the links cascade uniqueness and usages resolver.
    /// </para>
    /// <para></para>
    /// </summary>
```

```
    /// <seealso cref="LinksUniquenessResolver{TLinkAddress}"/>
    public class LinksCascadeUniquenessAndUsagesResolver<TLinkAddress> :
    ↪  LinksUniquenessResolver<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksCascadeUniquenessAndUsagesResolver"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksCascadeUniquenessAndUsagesResolver(ILinks<TLinkAddress> links) : base(links)
        ↪  { }

        /// <summary>
        /// <para>
        /// Resolves the address change conflict using the specified old link address.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="oldLinkAddress">
        /// <para>The old link address.</para>
        /// <para></para>
        /// </param>
        /// <param name="newLinkAddress">
        /// <para>The new link address.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress ResolveAddressChangeConflict(TLinkAddress
        ↪  oldLinkAddress, TLinkAddress newLinkAddress, WriteHandler<TLinkAddress>? handler)
        {
            var constants = _links.Constants;
            WriteHandlerState<TLinkAddress> handlerState = new(constants.Continue,
            ↪  constants.Break, handler);
            // Use Facade (the last decorator) to ensure recursion working correctly
            handlerState.Apply(_facade.MergeUsages(oldLinkAddress, newLinkAddress,
            ↪  handlerState.Handler));
            handlerState.Apply(base.ResolveAddressChangeConflict(oldLinkAddress, newLinkAddress,
            ↪  handlerState.Handler));
            return handlerState.Result;
        }
    }
}
```

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

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

#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<TLinkAddress> : LinksDecoratorBase<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksCascadeUsagesResolver"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
```

```
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksCascadeUsagesResolver(ILinks<TLinkAddress> links) : base(links) { }

        /// <summary>
        /// <para>
        /// Deletes the restriction.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restriction">
        /// <para>The restriction.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLinkAddress Delete(IList<TLinkAddress>? restriction,
        ↪  WriteHandler<TLinkAddress>? handler)
        {
            var constants = _links.Constants;
            WriteHandlerState<TLinkAddress> handlerState = new(constants.Continue,
            ↪  constants.Break, handler);
            var linkIndex = restriction[_constants.IndexPart];
            // Use Facade (the last decorator) to ensure recursion working correctly
            handlerState.Apply(_facade.DeleteAllUsages(linkIndex, handlerState.Handler));
            handlerState.Apply(_links.Delete(restriction, handlerState.Handler));
            return handlerState.Result;
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Decorators
{
    /// <summary>
    /// <para>
    /// Represents the links decorator base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
    /// <seealso cref="ILinks{TLinkAddress}"/>
    public abstract class LinksDecoratorBase<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
    ↪  ILinks<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// The constants.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly LinksConstants<TLinkAddress> _constants;

        /// <summary>
        /// <para>
        /// Gets the constants value.
        /// </para>
        /// <para></para>
        /// </summary>
        public LinksConstants<TLinkAddress> Constants
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _constants;
        }

        /// <summary>
        /// <para>
        /// The facade.
        /// </para>
        /// <para></para>
        /// </summary>
        protected ILinks<TLinkAddress> _facade;

```

```
        /// <summary>
        /// <para>
        /// Gets or sets the facade value.
        /// </para>
        /// <para></para>
        /// </summary>
        public ILinks<TLinkAddress> Facade
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _facade;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set
            {
                _facade = value;
                if (_links is LinksDecoratorBase<TLinkAddress> decorator)
                {
                    decorator.Facade = value;
                }
            }
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksDecoratorBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected LinksDecoratorBase(ILinks<TLinkAddress> links) : base(links)
        {
            _constants = links.Constants;
            Facade = this;
        }

        /// <summary>
        /// <para>
        /// Counts the restriction.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restriction">
        /// <para>The restriction.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLinkAddress Count(IList<TLinkAddress>? restriction) =>
        ↪  _links.Count(restriction);

        /// <summary>
        /// <para>
        /// Eaches the handler.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="handler">
        /// <para>The handler.</para>
        /// <para></para>
        /// </param>
        /// <param name="restriction">
        /// <para>The restriction.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLinkAddress Each(IList<TLinkAddress>? restriction,
        ↪  ReadHandler<TLinkAddress>? handler) => _links.Each(restriction, handler);

```

```
124         /// <summary>
125         /// <para>
126         /// Creates the restriction.
127         /// </para>
128         /// <para></para>
129         /// </summary>
130         /// <param name="restriction">
131         /// <para>The restriction.</para>
132         /// <para></para>
133         /// </param>
134         /// <returns>
135         /// <para>The link</para>
136         /// <para></para>
137         /// </returns>
138         [MethodImpl(MethodImplOptions.AggressiveInlining)]
139         public virtual TLinkAddress Create(IList<TLinkAddress>? substitution,
            ↪ WriteHandler<TLinkAddress>? handler) => _links.Create(substitution, handler);
140
141         /// <summary>
142         /// <para>
143         /// Updates the restriction.
144         /// </para>
145         /// <para></para>
146         /// </summary>
147         /// <param name="restriction">
148         /// <para>The restriction.</para>
149         /// <para></para>
150         /// </param>
151         /// <param name="substitution">
152         /// <para>The substitution.</para>
153         /// <para></para>
154         /// </param>
155         /// <returns>
156         /// <para>The link</para>
157         /// <para></para>
158         /// </returns>
159         [MethodImpl(MethodImplOptions.AggressiveInlining)]
160         public virtual TLinkAddress Update(IList<TLinkAddress>? restriction,
            ↪ IList<TLinkAddress>? substitution, WriteHandler<TLinkAddress>? handler) =>
            ↪ _links.Update(restriction, substitution, handler);
161
162         /// <summary>
163         /// <para>
164         /// Deletes the restriction.
165         /// </para>
166         /// <para></para>
167         /// </summary>
168         /// <param name="restriction">
169         /// <para>The restriction.</para>
170         /// <para></para>
171         /// </param>
172         [MethodImpl(MethodImplOptions.AggressiveInlining)]
173         public virtual TLinkAddress Delete(IList<TLinkAddress>? restriction,
            ↪ WriteHandler<TLinkAddress>? handler) => _links.Delete(restriction, handler);
174     }
175 }
```

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

```
1  using System.Runtime.CompilerServices;
2  using Platform.Disposables;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5  #pragma warning disable CA1063 // Implement IDisposable Correctly
6
7  namespace Platform.Data.Doublets.Decorators
8  {
9      /// <summary>
10     /// <para>
11     /// Represents the links disposable decorator base.
12     /// </para>
13     /// <para></para>
14     /// </summary>
15     /// <seealso cref="LinksDecoratorBase{TLinkAddress}"/>
16     /// <seealso cref="ILinks{TLinkAddress}"/>
17     /// <seealso cref="System.IDisposable"/>
18     public abstract class LinksDisposableDecoratorBase<TLinkAddress> :
        ↪ LinksDecoratorBase<TLinkAddress>, ILinks<TLinkAddress>, System.IDisposable
19     {
```

```
        /// <summary>
        /// <para>
        /// Represents the disposable with multiple calls allowed.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <seealso cref="Disposable"/>
        protected class DisposableWithMultipleCallsAllowed : Disposable
        {
            /// <summary>
            /// <para>
            /// Initializes a new <see cref="DisposableWithMultipleCallsAllowed"/> instance.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="disposal">
            /// <para>A disposal.</para>
            /// <para></para>
            /// </param>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public DisposableWithMultipleCallsAllowed(Disposal disposal) : base(disposal) { }

            /// <summary>
            /// <para>
            /// Gets the allow multiple dispose calls value.
            /// </para>
            /// <para></para>
            /// </summary>
            protected override bool AllowMultipleDisposeCalls
            {
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                get => true;
            }
        }

        /// <summary>
        /// <para>
        /// The disposable.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly DisposableWithMultipleCallsAllowed Disposable;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksDisposableDecoratorBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected LinksDisposableDecoratorBase(ILinks<TLinkAddress> links) : base(links) =>
        ↪  Disposable = new DisposableWithMultipleCallsAllowed(Dispose);

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

        /// <summary>
        /// <para>
        /// Disposes this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Dispose() => Disposable.Dispose();

        /// <summary>
        /// <para>
        /// Disposes the manual.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="manual">
        /// <para>The manual.</para>
        /// <para></para>
```

```
        /// </param>
        /// <param name="wasDisposed">
        /// <para>The was disposed.</para>
        /// <para></para>
        /// </param>
        [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;
using Platform.Delegates;

#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).
    /// <summary>
    /// <para>
    /// Represents the links inner reference existence validator.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksDecoratorBase{TLinkAddress}"/>
    public class LinksInnerReferenceExistenceValidator<TLinkAddress> : LinksDecoratorBase<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksInnerReferenceExistenceValidator"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksInnerReferenceExistenceValidator(ILinks<TLinkAddress> links) : base(links) { }

        /// <summary>
        /// <para>
        /// Eaches the handler.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="handler">
        /// <para>The handler.</para>
        /// <para></para>
        /// </param>
        /// <param name="restriction">
        /// <para>The restriction.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLinkAddress Each(IList<TLinkAddress>? restriction, ReadHandler<TLinkAddress>? handler)
        {
            var links = _links;
            links.EnsureInnerReferenceExists(restriction, nameof(restriction));
            return links.Each(restriction, handler);
        }
```

```

        /// <summary>
        /// <para>
        /// Updates the restriction.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restriction">
        /// <para>The restriction.</para>
        /// <para></para>
        /// </param>
        /// <param name="substitution">
        /// <para>The substitution.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLinkAddress Update(IList<TLinkAddress>? restriction,
        ↪  IList<TLinkAddress>? substitution, WriteHandler<TLinkAddress>? handler)
        {
            // TODO: Possible values: null, ExistentLink or NonExistentHybrid(ExternalReference)
            var links = _links;
            links.EnsureInnerReferenceExists(restriction, nameof(restriction));
            links.EnsureInnerReferenceExists(substitution, nameof(substitution));
            return links.Update(restriction, substitution, handler);
        }

        /// <summary>
        /// <para>
        /// Deletes the restriction.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restriction">
        /// <para>The restriction.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLinkAddress Delete(IList<TLinkAddress>? restriction,
        ↪  WriteHandler<TLinkAddress>? handler)
        {
            var link = restriction[_constants.IndexPart];
            var links = _links;
            links.EnsureLinkExists(link, nameof(link));
            return links.Delete(restriction, handler);
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Decorators
{
    /// <summary>
    /// <para>
    /// Represents the links itself constant to self reference resolver.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksDecoratorBase{TLinkAddress}"/>
    public class LinksItselfConstantToSelfReferenceResolver<TLinkAddress> :
    ↪  LinksDecoratorBase<TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksItselfConstantToSelfReferenceResolver"/> instance.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksItselfConstantToSelfReferenceResolver(ILinks<TLinkAddress> links) :
        ↪  base(links) { }

        /// <summary>
        /// <para>
        /// Eaches the handler.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="handler">
        /// <para>The handler.</para>
        /// <para></para>
        /// </param>
        /// <param name="restriction">
        /// <para>The restriction.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLinkAddress Each(IList<TLinkAddress>? restriction,
        ↪  ReadHandler<TLinkAddress>? handler)
        {
            var constants = _constants;
            var itselfConstant = constants.Itself;
            if (!_equalityComparer.Equals(constants.Any, itselfConstant) &&
            ↪  restriction.Contains(itselfConstant))
            {
                // Itself constant is not supported for Each method right now, skipping execution
                return constants.Continue;
            }
            return _links.Each(restriction, handler);
        }

        /// <summary>
        /// <para>
        /// Updates the restriction.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restriction">
        /// <para>The restriction.</para>
        /// <para></para>
        /// </param>
        /// <param name="substitution">
        /// <para>The substitution.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLinkAddress Update(IList<TLinkAddress>? restriction,
        ↪  IList<TLinkAddress>? substitution, WriteHandler<TLinkAddress>? handler) =>
        ↪  _links.Update(restriction, _links.ResolveConstantAsSelfReference(_constants.Itself,
        ↪  restriction, substitution), handler);
    }
}
```

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

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

#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<TLinkAddress> : LinksDecoratorBase<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksNonExistentDependenciesCreator"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksNonExistentDependenciesCreator(ILinks<TLinkAddress> links) : base(links) { }

        /// <summary>
        /// <para>
        /// Updates the restriction.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restriction">
        /// <para>The restriction.</para>
        /// <para></para>
        /// </param>
        /// <param name="substitution">
        /// <para>The substitution.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLinkAddress Update(IList<TLinkAddress>? restriction, IList<TLinkAddress>? substitution, WriteHandler<TLinkAddress>? handler)
        {
            var constants = _constants;
            var links = _links;
            links.EnsureCreated(substitution[constants.SourcePart], substitution[constants.TargetPart]);
            return links.Update(restriction, substitution, handler);
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Decorators
{
    /// <summary>
    /// <para>
    /// Represents the links null constant to self reference resolver.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksDecoratorBase{TLinkAddress}"/>
    public class LinksNullConstantToSelfReferenceResolver<TLinkAddress> : LinksDecoratorBase<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksNullConstantToSelfReferenceResolver"/> instance.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksNullConstantToSelfReferenceResolver(ILinks<TLinkAddress> links) :
        ↪ base(links) { }

        /// <summary>
        /// <para>
        /// Creates the substitution.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="substitution">
        /// <para>The substitution.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLinkAddress Create(IList<TLinkAddress>? substitution,
        ↪ WriteHandler<TLinkAddress>? handler)
        {
            return _links.CreatePoint(handler);
        }

        /// <summary>
        /// <para>
        /// Updates the substitution.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restriction">
        /// <para>The substitution.</para>
        /// <para></para>
        /// </param>
        /// <param name="substitution">
        /// <para>The substitution.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLinkAddress Update(IList<TLinkAddress>? restriction,
        ↪ IList<TLinkAddress>? substitution, WriteHandler<TLinkAddress>? handler) =>
        ↪ _links.Update(restriction, _links.ResolveConstantAsSelfReference(_constants.Null,
        ↪ restriction, substitution), handler);
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Decorators
{
    /// <summary>
    /// <para>
    /// Represents the links uniqueness resolver.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksDecoratorBase{TLinkAddress}"/>
    public class LinksUniquenessResolver<TLinkAddress> : LinksDecoratorBase<TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪ EqualityComparer<TLinkAddress>.Default;
```

```
20
21          /// <summary>
22          /// <para>
23          /// Initializes a new <see cref="LinksUniquenessResolver"/> instance.
24          /// </para>
25          /// <para></para>
26          /// </summary>
27          /// <param name="links">
28          /// <para>A links.</para>
29          /// <para></para>
30          /// </param>
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          public LinksUniquenessResolver(ILinks<TLinkAddress> links) : base(links) { }
33
34          /// <summary>
35          /// <para>
36          /// Updates the restriction.
37          /// </para>
38          /// <para></para>
39          /// </summary>
40          /// <param name="restriction">
41          /// <para>The restriction.</para>
42          /// <para></para>
43          /// </param>
44          /// <param name="substitution">
45          /// <para>The substitution.</para>
46          /// <para></para>
47          /// </param>
48          /// <returns>
49          /// <para>The link</para>
50          /// <para></para>
51          /// </returns>
52          [MethodImpl(MethodImplOptions.AggressiveInlining)]
53          public override TLinkAddress Update(IList<TLinkAddress>? restriction,
            ↪ IList<TLinkAddress>? substitution, WriteHandler<TLinkAddress>? handler)
54          {
55              var constants = _constants;
56              var links = _links;
57              var newLinkAddress = links.SearchOrDefault(substitution[constants.SourcePart],
                ↪ substitution[constants.TargetPart]);
58              if (_equalityComparer.Equals(newLinkAddress, default))
59              {
60                  return links.Update(restriction, substitution, handler);
61              }
62              return ResolveAddressChangeConflict(restriction[constants.IndexPart],
                ↪ newLinkAddress, handler);
63          }
64
65          /// <summary>
66          /// <para>
67          /// Resolves the address change conflict using the specified old link address.
68          /// </para>
69          /// <para></para>
70          /// </summary>
71          /// <param name="oldLinkAddress">
72          /// <para>The old link address.</para>
73          /// <para></para>
74          /// </param>
75          /// <param name="newLinkAddress">
76          /// <para>The new link address.</para>
77          /// <para></para>
78          /// </param>
79          /// <returns>
80          /// <para>The new link address.</para>
81          /// <para></para>
82          /// </returns>
83          [MethodImpl(MethodImplOptions.AggressiveInlining)]
84          protected virtual TLinkAddress ResolveAddressChangeConflict(TLinkAddress oldLinkAddress,
            ↪ TLinkAddress newLinkAddress, WriteHandler<TLinkAddress>? handler)
85          {
86              if (!_equalityComparer.Equals(oldLinkAddress, newLinkAddress) &&
                ↪ _links.Exists(oldLinkAddress))
87              {
88                  return _facade.Delete(oldLinkAddress, handler);
89              }
90              return _links.Constants.Continue;
91          }
```

```
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Decorators
{
    /// <summary>
    /// <para>
    /// Represents the links uniqueness validator.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksDecoratorBase{TLinkAddress}"/>
    public class LinksUniquenessValidator<TLinkAddress> : LinksDecoratorBase<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksUniquenessValidator"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksUniquenessValidator(ILinks<TLinkAddress> links) : base(links) { }

        /// <summary>
        /// <para>
        /// Updates the restriction.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restriction">
        /// <para>The restriction.</para>
        /// <para></para>
        /// </param>
        /// <param name="substitution">
        /// <para>The substitution.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLinkAddress Update(IList<TLinkAddress>? restriction,
        ↪  IList<TLinkAddress>? substitution, WriteHandler<TLinkAddress>? handler)
        {
            var links = _links;
            var constants = _constants;
            links.EnsureDoesNotExists(substitution[constants.SourcePart],
            ↪  substitution[constants.TargetPart]);
            return links.Update(restriction, substitution, handler);
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Decorators
{
    /// <summary>
    /// <para>
    /// Represents the links usages validator.
```

```
13      /// </para>
14      /// <para></para>
15      /// </summary>
16      /// <seealso cref="LinksDecoratorBase{TLinkAddress}"/>
17      public class LinksUsagesValidator<TLinkAddress> : LinksDecoratorBase<TLinkAddress>
18      {
19          /// <summary>
20          /// <para>
21          /// Initializes a new <see cref="LinksUsagesValidator"/> instance.
22          /// </para>
23          /// <para></para>
24          /// </summary>
25          /// <param name="links">
26          /// <para>A links.</para>
27          /// <para></para>
28          /// </param>
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          public LinksUsagesValidator(ILinks<TLinkAddress> links) : base(links) { }
31
32          /// <summary>
33          /// <para>
34          /// Updates the restriction.
35          /// </para>
36          /// <para></para>
37          /// </summary>
38          /// <param name="restriction">
39          /// <para>The restriction.</para>
40          /// <para></para>
41          /// </param>
42          /// <param name="substitution">
43          /// <para>The substitution.</para>
44          /// <para></para>
45          /// </param>
46          /// <returns>
47          /// <para>The link</para>
48          /// <para></para>
49          /// </returns>
50          [MethodImpl(MethodImplOptions.AggressiveInlining)]
51          public override TLinkAddress Update(IList<TLinkAddress>? restriction,
            ↪  IList<TLinkAddress>? substitution, WriteHandler<TLinkAddress>? handler)
52          {
53              var links = _links;
54              links.EnsureNoUsages(restriction[_constants.IndexPart]);
55              return links.Update(restriction, substitution, handler);
56          }
57
58          /// <summary>
59          /// <para>
60          /// Deletes the restriction.
61          /// </para>
62          /// <para></para>
63          /// </summary>
64          /// <param name="restriction">
65          /// <para>The restriction.</para>
66          /// <para></para>
67          /// </param>
68          [MethodImpl(MethodImplOptions.AggressiveInlining)]
69          public override TLinkAddress Delete(IList<TLinkAddress>? restriction,
            ↪  WriteHandler<TLinkAddress>? handler)
70          {
71              var link = restriction[_constants.IndexPart];
72              var links = _links;
73              links.EnsureNoUsages(link);
74              return links.Delete(restriction, handler);
75          }
76      }
77  }
```

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

```
1   using System.Collections.Generic;
2   using System.IO;
3   using Platform.Delegates;
4
5   namespace Platform.Data.Doublets.Decorators
6   {
7       public class LoggingDecorator<TLinkAddress> : LinksDecoratorBase<TLinkAddress>
8       {
9           private readonly Stream _logStream;
10          private readonly StreamWriter _logStreamWriter;
```

```
        public LoggingDecorator(ILinks<TLinkAddress> links, Stream logStream) : base(links)
        {
            _logStream = logStream;
            _logStreamWriter = new StreamWriter(_logStream);
            _logStreamWriter.AutoFlush = true;
        }

        public override TLinkAddress Create(IList<TLinkAddress>? substitution,
        ↪  WriteHandler<TLinkAddress>? handler)
        {
            WriteHandlerState<TLinkAddress> handlerState = new(_constants.Continue,
            ↪  _constants.Break, handler);
            return base.Create(substitution, (before, after) =>
            {
                if (handlerState.Handler != null)
                {
                    handlerState.Apply(handlerState.Handler(before, after));
                }
                _logStreamWriter.WriteLine($"Create. Before: {new Link<TLinkAddress>(before)}.
                ↪  After: {new Link<TLinkAddress>(after)}");
                return _constants.Continue;
            });
        }

        public override TLinkAddress Update(IList<TLinkAddress>? restriction,
        ↪  IList<TLinkAddress>? substitution, WriteHandler<TLinkAddress>? handler)
        {
            WriteHandlerState<TLinkAddress> handlerState = new(_constants.Continue,
            ↪  _constants.Break, handler);
            return base.Update(restriction, substitution, (before, after) =>
            {
                if (handlerState.Handler != null)
                {
                    handlerState.Apply(handlerState.Handler(before, after));
                }
                _logStreamWriter.WriteLine($"Update. Before: {new Link<TLinkAddress>(before)}.
                ↪  After: {new Link<TLinkAddress>(after)}");
                return _constants.Continue;
            });
        }

        public override TLinkAddress Delete(IList<TLinkAddress>? restriction,
        ↪  WriteHandler<TLinkAddress>? handler)
        {
            WriteHandlerState<TLinkAddress> handlerState = new(_constants.Continue,
            ↪  _constants.Break, handler);
            return base.Delete(restriction, (before, after) =>
            {
                if (handlerState.Handler != null)
                {
                    handlerState.Apply(handlerState.Handler(before, after));
                }
                _logStreamWriter.WriteLine($"Delete. Before: {new Link<TLinkAddress>(before)}.
                ↪  After: {new Link<TLinkAddress>(after)}");
                return _constants.Continue;
            });
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Decorators
{
    /// <summary>
    /// <para>
    /// Represents the non null contents link deletion resolver.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksDecoratorBase{TLinkAddress}"/>
```

```
    public class NonNullContentsLinkDeletionResolver<TLinkAddress> : LinksDecoratorBase<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="NonNullContentsLinkDeletionResolver"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public NonNullContentsLinkDeletionResolver(ILinks<TLinkAddress> links) : base(links) { }

        /// <summary>
        /// <para>
        /// Deletes the restriction.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restriction">
        /// <para>The restriction.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLinkAddress Delete(IList<TLinkAddress>? restriction,
            WriteHandler<TLinkAddress>? handler)
        {
            var linkIndex = restriction[_constants.IndexPart];
            var constants = _links.Constants;
            WriteHandlerState<TLinkAddress> handlerResult = new(constants.Continue,
                constants.Break, handler);
            handlerResult.Apply(_links.EnforceResetValues(linkIndex, handlerResult.Handler));
            handlerResult.Apply(_links.Delete(restriction, handlerResult.Handler));
            return handlerResult.Result;
        }
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Delegates;
using TLinkAddress = System.UInt32;

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

namespace Platform.Data.Doublets.Decorators
{
    /// <summary>
    /// <para>
    /// Represents the int 32 links.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksDisposableDecoratorBase{TLinkAddress}"/>
    public class UInt32Links : LinksDisposableDecoratorBase<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt32Links"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32Links(ILinks<TLinkAddress> links) : base(links) { }

        /// <summary>
        /// <para>
        /// Creates the substitution.
        /// </para>
        /// <para></para>
```

```
        /// </summary>
        /// <param name="substitution">
        /// <para>The substitution.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLinkAddress Create(IList<TLinkAddress>? substitution,
        ↪  WriteHandler<TLinkAddress>? handler) => _links.CreatePoint(handler);

        /// <summary>
        /// <para>
        /// Updates the substitution.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restriction">
        /// <para>The substitution.</para>
        /// <para></para>
        /// </param>
        /// <param name="substitution">
        /// <para>The substitution.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLinkAddress Update(IList<TLinkAddress>? restriction,
        ↪  IList<TLinkAddress>? substitution, WriteHandler<TLinkAddress>? handler)
        {
            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 = restriction[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)
                {
                    var source = newSource == itselfConstant ? updatedLink : newSource;
                    var target = newTarget == itselfConstant ? updatedLink : newTarget;
                    return links.Update(new Link<TLinkAddress>(updatedLink, source, target),
                    ↪  handler);
                }
                return _links.Constants.Continue;
            }
            else
            {
                return _facade.MergeAndDelete(updatedLink, existedLink, handler);
            }
        }

        /// <summary>
        /// <para>
        /// Deletes the substitution.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restriction">
        /// <para>The substitution.</para>
        /// <para></para>
```

```
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLinkAddress Delete(IList<TLinkAddress>? restriction,
        ↪  WriteHandler<TLinkAddress>? handler)
        {
            var linkIndex = restriction[_constants.IndexPart];
            var constants = _links.Constants;
            WriteHandlerState<TLinkAddress> handlerState = new(constants.Continue,
            ↪  constants.Break, handler);
            handlerState.Apply( _links.EnforceResetValues(linkIndex, handlerState.Handler));
            handlerState.Apply(_facade.DeleteAllUsages(linkIndex, handlerState.Handler));
            handlerState.Apply(_links.Delete(restriction, handlerState.Handler));
            return handlerState.Result;
        }
    }
}
```

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

```
using System.Collections.Generic;
using System.Net.Security;
using System.Runtime.CompilerServices;
using Platform.Delegates;
using TLinkAddress = System.UInt64;

#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<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt64Links"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64Links(ILinks<TLinkAddress> links) : base(links) { }

        /// <summary>
        /// <para>
        /// Creates the substitution.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="substitution">
        /// <para>The substitution.</para>
```

```
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The TLinkAddress</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLinkAddress Create(IList<TLinkAddress>? substitution,
        ↪  WriteHandler<TLinkAddress>? handler) => _links.CreatePoint(handler);

        /// <summary>
        /// <para>
        /// Updates the substitution.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restriction">
        /// <para>The substitution.</para>
        /// <para></para>
        /// </param>
        /// <param name="substitution">
        /// <para>The substitution.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The TLinkAddress</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLinkAddress Update(IList<TLinkAddress>? restriction,
        ↪  IList<TLinkAddress>? substitution, WriteHandler<TLinkAddress>? handler)
        {
            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 = restriction[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)
                {
                    var source = newSource == itselfConstant ? updatedLink : newSource;
                    var target = newTarget == itselfConstant ? updatedLink : newTarget;
                    return links.Update(new Link<TLinkAddress>(updatedLink, source, target),
                    ↪  handler);
                }
                return _links.Constants.Continue;
            }
            else
            {
                return _facade.MergeAndDelete(updatedLink, existedLink, handler);
            }
        }

        /// <summary>
        /// <para>
        /// Deletes the substitution.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restriction">
        /// <para>The substitution.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLinkAddress Delete(IList<TLinkAddress>? restriction,
        ↪  WriteHandler<TLinkAddress>? handler)
```

```
        {
            var linkIndex = restriction[_constants.IndexPart];
            var constants = _links.Constants;
            WriteHandlerState<TLinkAddress> handlerState = new(constants.Continue,
            ↪  constants.Break, handler);
            handlerState.Apply(_links.EnforceResetValues(linkIndex, handlerState.Handler));
            handlerState.Apply(_facade.DeleteAllUsages(linkIndex, handlerState.Handler));
            handlerState.Apply(_links.Delete(restriction, handlerState.Handler));
            return handlerState.Result;
        }
    }
}
```

1.17 ./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;
using Platform.Delegates;

#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<TLinkAddress> : LinksDecoratorBase<TLinkAddress>,
    ↪  IUniLinks<TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UniLinks"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        public UniLinks(ILinks<TLinkAddress> links) : base(links) { }
        private struct Transition
        {
            /// <summary>
            /// <para>
            /// The before.
            /// </para>
            /// <para></para>
            /// </summary>
            public IList<TLinkAddress>? Before;
            /// <summary>
            /// <para>
            /// The after.
            /// </para>
            /// <para></para>
            /// </summary>
            public IList<TLinkAddress>? After;

            /// <summary>
            /// <para>
            /// Initializes a new <see cref="Transition"/> instance.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="before">
            /// <para>A before.</para>
            /// <para></para>
            /// </param>
```

```
            /// <param name="after">
            /// <para>A after.</para>
            /// <para></para>
            /// </param>
            public Transition(IList<TLinkAddress>? before, IList<TLinkAddress>? after)
            {
                Before = before;
                After = after;
            }
        }

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

        // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution
        ↪  (Links-Expression)
        /// <summary>
        /// <para>
        /// Triggers the restriction.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restriction">
        /// <para>The restriction.</para>
        /// <para></para>
        /// </param>
        /// <param name="matchedHandler">
        /// <para>The matched handler.</para>
        /// <para></para>
        /// </param>
        /// <param name="substitution">
        /// <para>The substitution.</para>
        /// <para></para>
        /// </param>
        /// <param name="substitutedHandler">
        /// <para>The substituted handler.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public TLinkAddress Trigger(IList<TLinkAddress>? restriction,
        ↪  WriteHandler<TLinkAddress>? matchedHandler, IList<TLinkAddress>? substitution,
        ↪  WriteHandler<TLinkAddress>? 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);
```

```
131             ////                    var newValue = Memory.GetLinkValue(link);
132             ////                    newValue[Constants.IndexPart] = Constants.Itself;
133             ////                    newValue[Constants.SourcePart] =
                ↪  Equals(substitution[Constants.SourcePart], Constants.Itself) ?
                ↪  matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
134             ////                    newValue[Constants.TargetPart] =
                ↪  Equals(substitution[Constants.TargetPart], Constants.Itself) ?
                ↪  matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
135             ////                    var matchDecision = matchedHandler(matchedLink, newValue);
136             ////                    if (Equals(matchDecision, Constants.Break))
137             ////                        return false;
138             ////                    if (!Equals(matchDecision, Constants.Skip))
139             ////                        transitions.Add(new Transition(matchedLink, newValue));
140             ////                    return true;
141             ////                };
142             ////                if (!Memory.Each(handler, restriction))
143             ////                    return Constants.Break;
144             ////            }
145             ////        }
146             ////        else
147             ////        {
148             ////            Func<T, bool> handler = link =>
149             ////            {
150             ////                var matchedLink = Memory.GetLinkValue(link);
151             ////                var matchDecision = matchedHandler(matchedLink, matchedLink);
152             ////                return !Equals(matchDecision, Constants.Break);
153             ////            };
154             ////            if (!Memory.Each(handler, restriction))
155             ////                return Constants.Break;
156             ////        }
157             ////    }
158             ////    else
159             ////    {
160             ////        if (substitution != null)
161             ////        {
162             ////            transitions = new List<IList<T>>();
163             ////            Func<T, bool> handler = link =>
164             ////            {
165             ////                var matchedLink = Memory.GetLinkValue(link);
166             ////                transitions.Add(matchedLink);
167             ////                return true;
168             ////            };
169             ////            if (!Memory.Each(handler, restriction))
170             ////                return Constants.Break;
171             ////        }
172             ////        else
173             ////        {
174             ////            return Constants.Continue;
175             ////        }
176             ////    }
177             ////}
178             ////if (substitution != null)
179             ////{
180             ////    // Есть причина делать замену (запись)
181             ////    if (substitutedHandler != null)
182             ////    {
183             ////    }
184             ////    else
185             ////    {
186             ////    }
187             ////}
188             ////return Constants.Continue;
189
190             //if (restriction.IsNullOrEmpty()) // Create
191             //{
192             //    substitution[Constants.IndexPart] = Memory.AllocateLink();
193             //    Memory.SetLinkValue(substitution);
194             //}
195             //else if (substitution.IsNullOrEmpty()) // Delete
196             //{
197             //    Memory.FreeLink(restriction[Constants.IndexPart]);
198             //}
199             //else if (restriction.EqualTo(substitution)) // Read or ("repeat" the state) // Each
200             //{
201             //    // No need to collect links to list
202             //    // Skip == Continue
203             //    // No need to check substituedHandler
```

```
204             //    if (!Memory.Each(link => !Equals(matchedHandler(Memory.GetLinkValue(link)),
                ↪  Constants.Break), restriction))
205             //        return Constants.Break;
206             //}
207             //else // Update
208             //{
209             //    //List<IList<T>> matchedLinks = null;
210             //    if (matchedHandler != null)
211             //    {
212             //        matchedLinks = new List<IList<T>>();
213             //        Func<T, bool> handler = link =>
214             //        {
215             //            var matchedLink = Memory.GetLinkValue(link);
216             //            var matchDecision = matchedHandler(matchedLink);
217             //            if (Equals(matchDecision, Constants.Break))
218             //                return false;
219             //            if (!Equals(matchDecision, Constants.Skip))
220             //                matchedLinks.Add(matchedLink);
221             //            return true;
222             //        };
223             //        if (!Memory.Each(handler, restriction))
224             //            return Constants.Break;
225             //    }
226             //    if (!matchedLinks.IsNullOrEmpty())
227             //    {
228             //        var totalMatchedLinks = matchedLinks.Count;
229             //        for (var i = 0; i < totalMatchedLinks; i++)
230             //        {
231             //            var matchedLink = matchedLinks[i];
232             //            if (substitutedHandler != null)
233             //            {
234             //                var newValue = new List<T>(); // TODO: Prepare value to update here
235             //                // TODO: Decide is it actually needed to use Before and After
                ↪  substitution handling.
236             //                var substitutedDecision = substitutedHandler(matchedLink,
                ↪  newValue);
237             //                if (Equals(substitutedDecision, Constants.Break))
238             //                    return Constants.Break;
239             //                if (Equals(substitutedDecision, Constants.Continue))
240             //                {
241             //                    // Actual update here
242             //                    Memory.SetLinkValue(newValue);
243             //                }
244             //                if (Equals(substitutedDecision, Constants.Skip))
245             //                {
246             //                    // Cancel the update. TODO: decide use separate Cancel
                ↪  constant or Skip is enough?
247             //                }
248             //            }
249             //        }
250             //    }
251             //}
252             return _constants.Continue;
253         }
254 
255         /// <summary>
256         /// <para>
257         /// Triggers the pattern or condition.
258         /// </para>
259         /// <para></para>
260         /// </summary>
261         /// <param name="patternOrCondition">
262         /// <para>The pattern or condition.</para>
263         /// <para></para>
264         /// </param>
265         /// <param name="matchHandler">
266         /// <para>The match handler.</para>
267         /// <para></para>
268         /// </param>
269         /// <param name="substitution">
270         /// <para>The substitution.</para>
271         /// <para></para>
272         /// </param>
273         /// <param name="substitutionHandler">
274         /// <para>The substitution handler.</para>
275         /// <para></para>
276         /// </param>
```

```
277         /// <exception cref="NotImplementedException">
278         /// <para></para>
279         /// <para></para>
280         /// </exception>
281         /// <exception cref="NotSupportedException">
282         /// <para></para>
283         /// <para></para>
284         /// </exception>
285         /// <exception cref="NotSupportedException">
286         /// <para></para>
287         /// <para></para>
288         /// </exception>
289         /// <exception cref="NotSupportedException">
290         /// <para></para>
291         /// <para></para>
292         /// </exception>
293         /// <exception cref="NotSupportedException">
294         /// <para></para>
295         /// <para></para>
296         /// </exception>
297         /// <returns>
298         /// <para>The link</para>
299         /// <para></para>
300         /// </returns>
301         public TLinkAddress Trigger(IList<TLinkAddress>? patternOrCondition,
            ↪ ReadHandler<TLinkAddress>? matchHandler, IList<TLinkAddress>? substitution,
            ↪ WriteHandler<TLinkAddress>? substitutionHandler)
302         {
303             var constants = _constants;
304             if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
305             {
306                 return constants.Continue;
307             }
308             else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO:
                ↪ Check if it is a correct condition
309             {
310                 // Or it only applies to trigger without matchHandler.
311                 throw new NotImplementedException();
312             }
313             else if (!substitution.IsNullOrEmpty()) // Creation
314             {
315                 var before = Array.Empty<TLinkAddress>();
316                 // Что должно означать False здесь? Остановиться (перестать идти) или пропустить
                    ↪ (пройти мимо) или пустить (взять)?
317                 if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                    ↪ constants.Break))
318                 {
319                     return constants.Break;
320                 }
321                 var after = (IList<TLinkAddress>?)substitution.ToArray();
322                 if (_equalityComparer.Equals(after[0], default))
323                 {
324                     var newLink = _links.Create();
325                     after[0] = newLink;
326                 }
327                 if (substitution.Count == 1)
328                 {
329                     after = _links.GetLink(substitution[0]);
330                 }
331                 else if (substitution.Count == 3)
332                 {
333                     //Links.Create(after);
334                 }
335                 else
336                 {
337                     throw new NotSupportedException();
338                 }
339                 return matchHandler != null ? substitutionHandler(before, after) :
                    ↪ constants.Continue;
340             }
341             else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
342             {
343                 if (patternOrCondition.Count == 1)
344                 {
345                     var linkToDelete = patternOrCondition[0];
346                     var before = _links.GetLink(linkToDelete);
347                     if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                        ↪ constants.Break))
```

```
                {
                    return constants.Break;
                }
                var after = Array.Empty<TLinkAddress>();
                _links.Update(linkToDelete, constants.Null, constants.Null);
                _links.Delete(linkToDelete);
                return matchHandler != null ? substitutionHandler(before, after) :
                ↪  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<TLinkAddress>?)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();
                }
                return matchHandler != null ? substitutionHandler(before, after) :
                ↪  constants.Continue;
            }
            else
            {
                throw new NotSupportedException();
            }
        }
    }

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

```
422             return changes;
423         }
424         private TLinkAddress AlwaysContinue(IList<TLinkAddress>? linkToMatch) =>
            ↪ _constants.Continue;
425     }
426 }
```

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets
8  {
9
10     /// <summury>
11     /// <para>.</para>
12     /// <para>.</para>
13     /// </summury>
14     /// <typeparam>
15     /// <para>.</para>
16     /// <para>.</para>
17     /// </typeparam>
18     public struct Doublet<T> : IEquatable<Doublet<T>>
19     {
20         private static readonly EqualityComparer<T> _equalityComparer =
           ↪ EqualityComparer<T>.Default;
21
22         /// <summury>
23         /// <para>.</para>
24         /// <para>.</para>
25         /// </summury>
26         /// <typeparam name="T">
27         /// <para>.</para>
28         /// <para>.</para>
29         /// </typeparam>
30         public readonly T Source;
31
32         /// <summury>
33         /// <para>.</para>
34         /// <para>.</para>
35         /// </summury>
36         /// <typeparam name="T">
37         /// <para>.</para>
38         /// <para>.</para>
39         /// </typeparam>
40         public readonly T Target;
41
42         /// <summury>
43         /// <para>.</para>
44         /// <para>.</para>
45         /// </summury>
46         /// <typeparam name="T">
47         /// <para>.</para>
48         /// <para>.</para>
49         /// </typeparam>
50         /// <param name="source">
51         /// <para>.</para>
52         /// <para>.</para>
53         /// </param>
54         /// <param name="target">
55         /// <para>.</para>
56         /// <para>.</para>
57         /// </param>
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         public Doublet(T source, T target)
60         {
61             Source = source;
62             Target = target;
63         }
64
65         /// <summury>
66         /// <para>.</para>
67         /// <para>.</para>
68         /// </summury>
69         /// <returns>
70         /// <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);
    }
}
```

1.19 ./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>>
    {
        /// <summary>
        /// <para>
        /// The .
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly DoubletComparer<T> Default = new DoubletComparer<T>();

        /// <summary>
        /// <para>
        /// Determines whether this instance equals.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="x">
        /// <para>The .</para>
        /// <para></para>
        /// </param>
        /// <param name="y">
        /// <para>The .</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Equals(Doublet<T> x, Doublet<T> y) => x.Equals(y);

        /// <summary>
        /// <para>
        /// Gets the hash code using the specified obj.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="obj">
        /// <para>The obj.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The int</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public int GetHashCode(Doublet<T> obj) => obj.GetHashCode();
    }
```

```
}
```

1.20 ./csharp/Platform.Data.Doublets/FFI/UInt32UnitedMemoryLinks.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.InteropServices;
using Platform.Converters;
using Platform.Delegates;
using Platform.Disposables;

namespace Platform.Data.Doublets.FFI
{
    using TLinkAddress = System.UInt32;

    public class UInt32UnitedMemoryLinks : DisposableBase, ILinks<TLinkAddress>
    {
        public LinksConstants<TLinkAddress> Constants { get; }

        private readonly unsafe void* _ptr;

        public UInt32UnitedMemoryLinks(string path)
        {
            unsafe
            {
                _ptr = Methods.UInt32UnitedMemoryLinks_New(path);

                // TODO: Update api
                Constants = new LinksConstants<TLinkAddress>(enableExternalReferencesSupport: true);
            }
        }

        public TLinkAddress Count(IList<TLinkAddress>? restriction)
        {
            unsafe
            {
                var array = stackalloc uint[restriction.Count];
                for (var i = 0; i < restriction.Count; i++)
                {
                    array[i] = restriction[i];
                }
                return Methods.UInt32UnitedMemoryLinks_Count(_ptr, array, (nuint)(restriction?.Count ?? 0));
            }
        }

        public TLinkAddress Each(IList<TLinkAddress>? restriction, ReadHandler<TLinkAddress>? handler)
        {
            unsafe
            {
                Methods.EachCallback_UInt32 callback = (link) => handler?.Invoke(new Link<TLinkAddress>(link.Index, link.Source, link.Target)) ?? Constants.Continue;
                var array = stackalloc uint[restriction.Count];
                for (var i = 0; i < restriction.Count; i++)
                {
                    array[i] = restriction[i];
                }
                return Methods.UInt32UnitedMemoryLinks_Each(_ptr, array, (nuint)(restriction?.Count ?? 0), callback);
            }
        }

        public TLinkAddress Create(IList<TLinkAddress>? substitution, WriteHandler<TLinkAddress>? handler)
        {
            unsafe
            {
                Methods.CreateCallback_UInt32 callback = (before, after) => handler?.Invoke(new Link<TLinkAddress>(before.Index, before.Source, before.Target), new Link<TLinkAddress>(after.Index, after.Source, after.Target)) ?? Constants.Continue;
                fixed (uint* substitutionPtr = (uint[])substitution)
                {
                    return Methods.UInt32UnitedMemoryLinks_Create(_ptr, substitutionPtr, (nuint)(substitution?.Count ?? 0), callback);
                }
```

```
            }
        }

        public TLinkAddress Update(IList<TLinkAddress>? restriction, IList<TLinkAddress>?
        ↪  substitution, WriteHandler<TLinkAddress>? handler)
        {
            unsafe
            {
                var restrictionArray = stackalloc uint[restriction.Count];
                for (var i = 0; i < restriction.Count; i++)
                {
                    restrictionArray[i] = restriction[i];
                }
                var substitutionArray = stackalloc uint[substitution.Count];
                for (var i = 0; i < restriction.Count; i++)
                {
                    substitutionArray[i] = restriction[i];
                }
                Methods.UpdateCallback_UInt32 callback = (before, after) => handler?.Invoke(new
                ↪  Link<TLinkAddress>(before.Index, before.Source, before.Target), new
                ↪  Link<TLinkAddress>(after.Index, after.Source, after.Target)) ??
                ↪  Constants.Continue;
                return Methods.UInt32UnitedMemoryLinks_Update(_ptr, restrictionArray,
                ↪  (nuint)(restriction?.Count ?? 0), substitutionArray,
                ↪  (nuint)(substitution?.Count ?? 0), callback);
            }
        }

        public TLinkAddress Delete(IList<TLinkAddress>? restriction, WriteHandler<TLinkAddress>?
        ↪  handler)
        {
            unsafe
            {
                var restrictionArray = stackalloc uint[restriction.Count];
                for (var i = 0; i < restriction.Count; i++)
                {
                    restrictionArray[i] = restriction[i];
                }
                Methods.DeleteCallback_UInt32 callback = (before, after) => handler?.Invoke(new
                ↪  Link<TLinkAddress>(before.Index, before.Source, before.Target), new
                ↪  Link<TLinkAddress>(after.Index, after.Source, after.Target)) ??
                ↪  Constants.Continue;
                return Methods.UInt32UnitedMemoryLinks_Delete(_ptr, restrictionArray,
                ↪  (nuint)(restriction?.Count ?? 0), callback);
            }
        }

        protected override void Dispose(bool manual, bool wasDisposed)
        {
            unsafe
            {
                if (wasDisposed || _ptr == null)
                {
                    return;
                }
                Methods.UInt32UnitedMemoryLinks_Drop(_ptr);
            }
        }
    }
}
```

1.21 ./csharp/Platform.Data.Doublets/FFI/UnitedMemoryLinks.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.InteropServices;
using Platform.Converters;
using Platform.Delegates;
using Platform.Disposables;

namespace Platform.Data.Doublets.FFI
{
    struct FfiLink_UInt8
    {
        public Byte Index;
        public Byte Source;
        public Byte Target;
    }

```

```
18      struct FfiLink_UInt16
19      {
20          public UInt16 Index;
21          public UInt16 Source;
22          public UInt16 Target;
23      }
24
25      struct FfiLink_UInt32
26      {
27          public UInt32 Index;
28          public UInt32 Source;
29          public UInt32 Target;
30      }
31
32      struct FfiLink_UInt64
33      {
34          public UInt64 Index;
35          public UInt64 Source;
36          public UInt64 Target;
37      }
38
39      unsafe static class Methods
40      {
41          private const string DllName = "Platform.Doublets";
42
43          [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
44          public delegate Byte EachCallback_UInt8(FfiLink_UInt8 link);
45
46          [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
47          public delegate UInt16 EachCallback_UInt16(FfiLink_UInt16 link);
48
49          [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
50          public delegate UInt32 EachCallback_UInt32(FfiLink_UInt32 link);
51
52          [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
53          public delegate UInt64 EachCallback_UInt64(FfiLink_UInt64 link);
54
55          [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
56          public delegate Byte CreateCallback_UInt8(FfiLink_UInt8 before, FfiLink_UInt8 after);
57
58          [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
59          public delegate UInt16 CreateCallback_UInt16(FfiLink_UInt16 before, FfiLink_UInt16
            ↪  after);
60
61          [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
62          public delegate UInt32 CreateCallback_UInt32(FfiLink_UInt32 before, FfiLink_UInt32
            ↪  after);
63
64          [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
65          public delegate UInt64 CreateCallback_UInt64(FfiLink_UInt64 before, FfiLink_UInt64
            ↪  after);
66
67          [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
68          public delegate Byte UpdateCallback_UInt8(FfiLink_UInt8 before, FfiLink_UInt8 after);
69
70          [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
71          public delegate UInt16 UpdateCallback_UInt16(FfiLink_UInt16 before, FfiLink_UInt16
            ↪  after);
72
73          [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
74          public delegate UInt32 UpdateCallback_UInt32(FfiLink_UInt32 before, FfiLink_UInt32
            ↪  after);
75
76          [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
77          public delegate UInt64 UpdateCallback_UInt64(FfiLink_UInt64 before, FfiLink_UInt64
            ↪  after);
78
79          [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
80          public delegate Byte DeleteCallback_UInt8(FfiLink_UInt8 before, FfiLink_UInt8 after);
81
82          [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
83          public delegate UInt16 DeleteCallback_UInt16(FfiLink_UInt16 before, FfiLink_UInt16
            ↪  after);
84
85          [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
86          public delegate UInt32 DeleteCallback_UInt32(FfiLink_UInt32 before, FfiLink_UInt32
            ↪  after);
87
88          [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
```

```
89          public delegate UInt64 DeleteCallback_UInt64(FfiLink_UInt64 before, FfiLink_UInt64
            ↪  after);
90
91          [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
92          public static extern void* ByteUnitedMemoryLinks_New(string path);
93
94          [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
95          public static extern void* UInt16UnitedMemoryLinks_New(string path);
96
97          [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
98          public static extern void* UInt32UnitedMemoryLinks_New(string path);
99
100         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
101         public static extern void* UInt64UnitedMemoryLinks_New(string path);
102
103         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
104         public static extern void ByteUnitedMemoryLinks_Drop(void* self);
105
106         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
107         public static extern void UInt16UnitedMemoryLinks_Drop(void* self);
108
109         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
110         public static extern void UInt32UnitedMemoryLinks_Drop(void* self);
111
112         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
113         public static extern void UInt64UnitedMemoryLinks_Drop(void* self);
114
115         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
116         public static extern byte ByteUnitedMemoryLinks_Create(void* self, byte* substitution,
            ↪  nuint substitutionLength, CreateCallback_UInt8 callback);
117
118         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
119         public static extern ushort UInt16UnitedMemoryLinks_Create(void* self, ushort*
            ↪  substitution, nuint substitutionLength, CreateCallback_UInt16 callback);
120
121         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
122         public static extern uint UInt32UnitedMemoryLinks_Create(void* self, uint* substitution,
            ↪  nuint substitutionLength, CreateCallback_UInt32 callback);
123
124         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
125         public static extern ulong UInt64UnitedMemoryLinks_Create(void* self, ulong*
            ↪  substitution, nuint substitutionLength, CreateCallback_UInt64 callback);
126
127         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
128         public static extern byte ByteUnitedMemoryLinks_Count(void* self, byte* restriction,
            ↪  nuint len);
129
130         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
131         public static extern ushort UInt16UnitedMemoryLinks_Count(void* self, ushort*
            ↪  restriction, nuint len);
132
133         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
134         public static extern uint UInt32UnitedMemoryLinks_Count(void* self, uint* restriction,
            ↪  nuint len);
135
136         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
137         public static extern ulong UInt64UnitedMemoryLinks_Count(void* self, ulong* restriction,
            ↪  nuint len);
138
139         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
140         public static extern byte ByteUnitedMemoryLinks_Each(void* self, byte* restriction,
            ↪  nuint len, EachCallback_UInt8 callback);
141
142         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
143         public static extern ushort UInt16UnitedMemoryLinks_Each(void* self, ushort*
            ↪  restriction, nuint len, EachCallback_UInt16 callback);
144
145         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
146         public static extern uint UInt32UnitedMemoryLinks_Each(void* self, uint* restriction,
            ↪  nuint len, EachCallback_UInt32 callback);
147
148         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
149         public static extern ulong UInt64UnitedMemoryLinks_Each(void* self, ulong* restriction,
            ↪  nuint len, EachCallback_UInt64 callback);
150
151         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
```

```
152         public static extern byte ByteUnitedMemoryLinks_Update(void* self, byte* restriction,
            ↪  nuint restrictionLength,  byte* substitution, nuint substitutionLength,
            ↪  UpdateCallback_UInt8 callback);
153
154         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
155         public static extern ushort UInt16UnitedMemoryLinks_Update(void* self, ushort*
            ↪  restriction, nuint restrictionLength,  ushort* substitution, nuint
            ↪  substitutionLength, UpdateCallback_UInt16 callback);
156
157         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
158         public static extern uint UInt32UnitedMemoryLinks_Update(void* self, uint* restriction,
            ↪  nuint restrictionLength,  uint* substitution, nuint substitutionLength,
            ↪  UpdateCallback_UInt32 callback);
159
160         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
161         public static extern ulong UInt64UnitedMemoryLinks_Update(void* self, ulong*
            ↪  restriction, nuint restrictionLength,  ulong* substitution, nuint
            ↪  substitutionLength, UpdateCallback_UInt64 callback);
162
163         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
164         public static extern byte ByteUnitedMemoryLinks_Delete(void* self, byte* restriction,
            ↪  nuint restrictionLength, DeleteCallback_UInt8 callback);
165
166         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
167         public static extern ushort UInt16UnitedMemoryLinks_Delete(void* self, ushort*
            ↪  restriction, nuint len, DeleteCallback_UInt16 callback);
168
169         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
170         public static extern uint UInt32UnitedMemoryLinks_Delete(void* self, uint* restriction,
            ↪  nuint len, DeleteCallback_UInt32 callback);
171
172         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
173         public static extern ulong UInt64UnitedMemoryLinks_Delete(void* self, ulong*
            ↪  restriction, nuint len, DeleteCallback_UInt64 callback);
174     }
175
176     public class UnitedMemoryLinks<TLinkAddress> : DisposableBase, ILinks<TLinkAddress> where
        ↪  TLinkAddress : struct
177     {
178         private static readonly UncheckedConverter<byte, TLinkAddress> from_u8 =
            ↪  UncheckedConverter<byte, TLinkAddress>.Default;
179         private static readonly UncheckedConverter<ushort, TLinkAddress> from_u16 =
            ↪  UncheckedConverter<ushort, TLinkAddress>.Default;
180         private static readonly UncheckedConverter<uint, TLinkAddress> from_u32 =
            ↪  UncheckedConverter<uint, TLinkAddress>.Default;
181         private static readonly UncheckedConverter<ulong, TLinkAddress> from_u64 =
            ↪  UncheckedConverter<ulong, TLinkAddress>.Default;
182         private static readonly UncheckedConverter<TLinkAddress, ulong> from_t =
            ↪  UncheckedConverter<TLinkAddress, ulong>.Default;
183
184         public LinksConstants<TLinkAddress> Constants { get; }
185
186         private readonly unsafe void* _ptr;
187
188         public UnitedMemoryLinks(string path)
189         {
190             TLinkAddress t = default;
191             unsafe
192             {
193                 _ptr = t switch
194                 {
195                     byte => Methods.ByteUnitedMemoryLinks_New(path),
196                     ushort => Methods.UInt16UnitedMemoryLinks_New(path),
197                     uint => Methods.UInt32UnitedMemoryLinks_New(path),
198                     ulong => Methods.UInt64UnitedMemoryLinks_New(path),
199                     _ => throw new NotImplementedException()
200                 };
201
202                 // TODO: Update api
203                 Constants = new LinksConstants<TLinkAddress>(enableExternalReferencesSupport:
                    ↪  true);
204             }
205         }
206
207         public TLinkAddress Count(IList<TLinkAddress>? restriction)
208         {
209             var restrictionLength = restriction?.Count ?? 0;
210             unsafe
211             {
```

```
212                 TLinkAddress t = default;
213                 switch (t)
214                 {
215                     case byte:
216                     {
217                         var restrictionArray = stackalloc byte[restrictionLength];
218                         var byteRestrictionArray = (IList<byte>)restriction;
219                         for (var i = 0; i < restrictionLength; i++)
220                         {
221                             restrictionArray[i] = byteRestrictionArray[i];
222                         }
223                         return from_u8.Convert(Methods.ByteUnitedMemoryLinks_Count(_ptr,
                            ↪ restrictionArray, (nuint)restrictionLength));
224                     }
225                     case ushort:
226                     {
227                         var restrictionArray = stackalloc ushort[restrictionLength];
228                         var ushortRestrictionArray = (IList<ushort>)restriction;
229                         for (var i = 0; i < restrictionLength; i++)
230                         {
231                             restrictionArray[i] = ushortRestrictionArray[i];
232                         }
233                         return from_u16.Convert(Methods.UInt16UnitedMemoryLinks_Count(_ptr,
                            ↪ restrictionArray, (nuint)restrictionLength));
234                     }
235                     case uint:
236                     {
237                         var restrictionArray = stackalloc uint[restrictionLength];
238                         var uintRestrictionArray = (IList<uint>)restriction;
239                         for (var i = 0; i < restrictionLength; i++)
240                         {
241                             restrictionArray[i] = uintRestrictionArray[i];
242                         }
243                         return from_u32.Convert(Methods.UInt32UnitedMemoryLinks_Count(_ptr,
                            ↪ restrictionArray, (nuint)restrictionLength));
244                     }
245                     case ulong:
246                     {
247                         {
248                             var restrictionArray = stackalloc ulong[restrictionLength];
249                             var ulongRestrictionArray = (IList<ulong>)restriction;
250                             for (var i = 0; i < restrictionLength; i++)
251                             {
252                                 restrictionArray[i] = ulongRestrictionArray[i];
253                             }
254                             return from_u64.Convert(Methods.UInt64UnitedMemoryLinks_Count(_ptr,
                                ↪ restrictionArray, (nuint)restrictionLength));
255                         }
256                     }
257                     default:
258                     {
259                         throw new NotImplementedException();
260                     }
261                 }
262             }
263         }
264
265         public TLinkAddress Each(IList<TLinkAddress>? restriction, ReadHandler<TLinkAddress>?
            ↪ handler)
266         {
267             var restrictionLength = restriction?.Count ?? 0;
268             unsafe
269             {
270                 TLinkAddress t = default;
271                 switch (t)
272                 {
273                     case byte:
274                     {
275                         byte Callback(FfiLink_UInt8 link) =>
                                ↪ (byte)from_t.Convert(handler?.Invoke(new
                                ↪ Link<TLinkAddress>(from_u8.Convert(link.Index),
                                ↪ from_u8.Convert(link.Source), from_u8.Convert(link.Target))) ??
                                ↪ Constants.Continue);
276                         var restrictionArray = stackalloc byte[restrictionLength];
277                         var byteRestrictionArray = (IList<byte>)restriction;
278                         for (var i = 0; i < restrictionLength; i++)
279                         {
280                             restrictionArray[i] = byteRestrictionArray[i];
```

```
                    }
                    return from_u8.Convert(Methods.ByteUnitedMemoryLinks_Each(_ptr,
                    ↪ restrictionArray, (nuint)restrictionLength, Callback));
                }
                case ushort:
                {
                    ushort Callback(FfiLink_UInt16 link) =>
                    ↪ (ushort)from_t.Convert(handler?.Invoke(new
                    ↪ Link<TLinkAddress>(from_u16.Convert(link.Index),
                    ↪ from_u16.Convert(link.Source), from_u16.Convert(link.Target))) ??
                    ↪ Constants.Continue);
                    var restrictionArray = stackalloc ushort[restrictionLength];
                    var ushortRestrictionArray = (IList<ushort>)restriction;
                    for (var i = 0; i < restrictionLength; i++)
                    {
                        restrictionArray[i] = ushortRestrictionArray[i];
                    }
                    return from_u16.Convert(Methods.UInt16UnitedMemoryLinks_Each(_ptr,
                    ↪ restrictionArray, (nuint)restrictionLength, Callback));
                }
                case uint:
                {
                    uint Callback(FfiLink_UInt32 link) =>
                    ↪ (uint)from_t.Convert(handler?.Invoke(new
                    ↪ Link<TLinkAddress>(from_u32.Convert(link.Index),
                    ↪ from_u32.Convert(link.Source), from_u32.Convert(link.Target))) ??
                    ↪ Constants.Continue);
                    var restrictionArray = stackalloc uint[restrictionLength];
                    var uintRestrictionArray = (IList<uint>)restriction;
                    for (var i = 0; i < restrictionLength; i++)
                    {
                        restrictionArray[i] = uintRestrictionArray[i];
                    }
                    return from_u32.Convert(Methods.UInt32UnitedMemoryLinks_Each(_ptr,
                    ↪ restrictionArray, (nuint)restrictionLength, Callback));
                }
                case ulong:
                {
                    {
                        ulong Callback(FfiLink_UInt64 link) =>
                        ↪ from_t.Convert(handler?.Invoke(new
                        ↪ Link<TLinkAddress>(from_u64.Convert(link.Index),
                        ↪ from_u64.Convert(link.Source), from_u64.Convert(link.Target)))
                        ↪ ?? Constants.Continue);
                        var restrictionArray = stackalloc UInt64[restrictionLength];
                        var ulongRestrictionArray = (IList<ulong>)restriction;
                        for (var i = 0; i < restrictionLength; i++)
                        {
                            restrictionArray[i] = ulongRestrictionArray[i];
                        }
                        return from_u64.Convert(Methods.UInt64UnitedMemoryLinks_Each(_ptr,
                        ↪ restrictionArray, (nuint)restrictionLength, Callback));
                    }
                }
                default:
                {
                    throw new NotImplementedException();
                }
            }
        }
    }

    public TLinkAddress Create(IList<TLinkAddress>? substitution,
    ↪ WriteHandler<TLinkAddress>? handler)
    {
        var substitutionLength = substitution?.Count ?? 0;
        unsafe
        {
            TLinkAddress t = default;
            switch (t)
            {
                case byte:
                {
```

```
337                         byte Callback(FfiLink_UInt8 before, FfiLink_UInt8 after) =>
                                (byte)from_t.Convert(handler?.Invoke(new
                            ↪   Link<TLinkAddress>(from_u8.Convert(before.Index),
                            ↪   from_u8.Convert(before.Source), from_u8.Convert(before.Target)), new
                            ↪   Link<TLinkAddress>(from_u8.Convert(after.Index),
                            ↪   from_u8.Convert(after.Source), from_u8.Convert(after.Target))) ??
                            ↪   Constants.Continue);
338                         var substitutionArray = stackalloc byte[substitutionLength];
339                         var byteSubstitutionArray = (IList<byte>)substitution;
340                         for (var i = 0; i < substitutionLength; i++)
341                         {
342                             substitutionArray[i] = byteSubstitutionArray[i];
343                         }
344                         return from_u8.Convert(Methods.ByteUnitedMemoryLinks_Create(_ptr,
                            ↪   substitutionArray, (nuint)(substitution?.Count ?? 0), Callback));
345                     }
346                     case ushort:
347                     {
348                         ushort Callback(FfiLink_UInt16 before, FfiLink_UInt16 after) =>
                                (ushort)from_t.Convert(handler?.Invoke(new
                            ↪   Link<TLinkAddress>(from_u16.Convert(before.Index),
                            ↪   from_u16.Convert(before.Source), from_u16.Convert(before.Target)),
                            ↪   new Link<TLinkAddress>(from_u16.Convert(after.Index),
                            ↪   from_u16.Convert(after.Source), from_u16.Convert(after.Target))) ??
                            ↪   Constants.Continue);
349                         var substitutionArray = stackalloc ushort[substitutionLength];
350                         var ushortSubstitutionArray = (IList<ushort>)substitution;
351                         for (var i = 0; i < substitutionLength; i++)
352                         {
353                             substitutionArray[i] = ushortSubstitutionArray[i];
354                         }
355                         return from_u16.Convert(Methods.UInt16UnitedMemoryLinks_Create(_ptr,
                            ↪   substitutionArray, (nuint)(substitution?.Count ?? 0), Callback));
356                     }
357                     case uint:
358                     {
359                         uint Callback(FfiLink_UInt32 before, FfiLink_UInt32 after) =>
                                (uint)from_t.Convert(handler?.Invoke(new
                            ↪   Link<TLinkAddress>(from_u32.Convert(before.Index),
                            ↪   from_u32.Convert(before.Source), from_u32.Convert(before.Target)),
                            ↪   new Link<TLinkAddress>(from_u32.Convert(after.Index),
                            ↪   from_u32.Convert(after.Source), from_u32.Convert(after.Target))) ??
                            ↪   Constants.Continue);
360                         var substitutionArray = stackalloc uint[substitutionLength];
361                         var uintSubstitutionArray = (IList<uint>)substitution;
362                         for (var i = 0; i < substitutionLength; i++)
363                         {
364                             substitutionArray[i] = uintSubstitutionArray[i];
365                         }
366                         return from_u32.Convert(Methods.UInt32UnitedMemoryLinks_Create(_ptr,
                            ↪   substitutionArray, (nuint)(substitution?.Count ?? 0), Callback));
367
368                     }
369                     case ulong:
370                     {
371                         ulong Callback(FfiLink_UInt64 before, FfiLink_UInt64 after) =>
                                (ulong)from_t.Convert(handler?.Invoke(new
                            ↪   Link<TLinkAddress>(from_u64.Convert(before.Index),
                            ↪   from_u64.Convert(before.Source), from_u64.Convert(before.Target)),
                            ↪   new Link<TLinkAddress>(from_u64.Convert(after.Index),
                            ↪   from_u64.Convert(after.Source), from_u64.Convert(after.Target))) ??
                            ↪   Constants.Continue);
372                         var substitutionArray = stackalloc ulong[substitutionLength];
373                         var ulongSubstitutionArray = (IList<ulong>)substitution;
374                         for (var i = 0; i < substitutionLength; i++)
375                         {
376                             substitutionArray[i] = ulongSubstitutionArray[i];
377                         }
378                         return from_u64.Convert(Methods.UInt64UnitedMemoryLinks_Create(_ptr,
                            ↪   substitutionArray, (nuint)(substitution?.Count ?? 0), Callback));
379
380                     }
381                     default:
382                     {
383                         throw new NotImplementedException();
384                     }
385                 };
386             }
```

```
387         }
388
389         public TLinkAddress Update(IList<TLinkAddress>? restriction, IList<TLinkAddress>?
            ↪ substitution, WriteHandler<TLinkAddress>? handler)
390         {
391             var restrictionLength = restriction?.Count ?? 0;
392             var substitutionLength = substitution?.Count ?? 0;
393             unsafe
394             {
395                 TLinkAddress t = default;
396                 switch (t)
397                 {
398                     case byte:
399                     {
400                         var restrictionArray = stackalloc byte[restrictionLength];
401                         var byteRestrictionArray = (IList<byte>)restriction;
402                         for (var i = 0; i < restrictionLength; i++)
403                         {
404                             restrictionArray[i] = byteRestrictionArray[i];
405                         }
406                         var substitutionArray = stackalloc byte[substitutionLength];
407                         var byteSubstitutionArray = (IList<byte>)substitution;
408                         for (var i = 0; i < substitutionLength; i++)
409                         {
410                             substitutionArray[i] = byteSubstitutionArray[i];
411                         }
412                         byte Callback(FfiLink_UInt8 before, FfiLink_UInt8 after) =>
                            ↪ (byte)from_t.Convert(handler?.Invoke(new
                            ↪ Link<TLinkAddress>(from_u8.Convert(before.Index),
                            ↪ from_u8.Convert(before.Source), from_u8.Convert(before.Target)), new
                            ↪ Link<TLinkAddress>(from_u8.Convert(after.Index),
                            ↪ from_u8.Convert(after.Source), from_u8.Convert(after.Target))) ??
                            ↪ Constants.Continue);
413                         return from_u8.Convert(Methods.ByteUnitedMemoryLinks_Update(_ptr,
                            ↪ restrictionArray, (nuint)restrictionLength, substitutionArray,
                            ↪ (nuint)(substitution?.Count ?? 0), Callback));
414                     }
415                     case ushort:
416                     {
417                         var restrictionArray = stackalloc ushort[restrictionLength];
418                         var ushortRestrictionArray = (IList<ushort>)restriction;
419                         for (var i = 0; i < restrictionLength; i++)
420                         {
421                             restrictionArray[i] = ushortRestrictionArray[i];
422                         }
423                         var substitutionArray = stackalloc ushort[substitutionLength];
424                         var ushortSubstitutionArray = (IList<ushort>)substitution;
425                         for (var i = 0; i < substitutionLength; i++)
426                         {
427                             substitutionArray[i] = ushortSubstitutionArray[i];
428                         }
429                         ushort Callback(FfiLink_UInt16 before, FfiLink_UInt16 after) =>
                            ↪ (ushort)from_t.Convert(handler?.Invoke(new
                            ↪ Link<TLinkAddress>(from_u16.Convert(before.Index),
                            ↪ from_u16.Convert(before.Source), from_u16.Convert(before.Target)),
                            ↪ new Link<TLinkAddress>(from_u16.Convert(after.Index),
                            ↪ from_u16.Convert(after.Source), from_u16.Convert(after.Target))) ??
                            ↪ Constants.Continue);
430                         return from_u16.Convert(Methods.UInt16UnitedMemoryLinks_Update(_ptr,
                            ↪ restrictionArray, (nuint)restrictionLength, substitutionArray,
                            ↪ (nuint)(substitution?.Count ?? 0), Callback));
431                     }
432                     case uint:
433                     {
434                         var restrictionArray = stackalloc uint[restrictionLength];
435                         var uintRestrictionArray = (IList<uint>)restriction;
436                         for (var i = 0; i < restrictionLength; i++)
437                         {
438                             restrictionArray[i] = uintRestrictionArray[i];
439                         }
440                         var substitutionArray = stackalloc uint[substitutionLength];
441                         var uintSubstitutionArray = (IList<uint>)substitution;
442                         for (var i = 0; i < substitutionLength; i++)
443                         {
444                             substitutionArray[i] = uintSubstitutionArray[i];
445                         }
```

```
446                         uint Callback(FfiLink_UInt32 before, FfiLink_UInt32 after) =>
                                (uint)from_t.Convert(handler?.Invoke(new
                            ↪   Link<TLinkAddress>(from_u32.Convert(before.Index),
                            ↪   from_u32.Convert(before.Source), from_u32.Convert(before.Target)),
                            ↪   new Link<TLinkAddress>(from_u32.Convert(after.Index),
                            ↪   from_u32.Convert(after.Source), from_u32.Convert(after.Target))) ??
                            ↪   Constants.Continue);
447                         return from_u32.Convert(Methods.UInt32UnitedMemoryLinks_Update(_ptr,
                            ↪   restrictionArray, (nuint)restrictionLength, substitutionArray,
                            ↪   (nuint)(substitution?.Count ?? 0), Callback));
448                     }
449                     case ulong:
450                     {
451                         var restrictionArray = stackalloc ulong[restrictionLength];
452                         var ulongRestrictionArray = (IList<ulong>)restriction;
453                         for (var i = 0; i < restrictionLength; i++)
454                         {
455                             restrictionArray[i] = ulongRestrictionArray[i];
456                         }
457                         var substitutionArray = stackalloc ulong[substitutionLength];
458                         var ulongSubstitutionArray = (IList<ulong>)substitution;
459                         for (var i = 0; i < substitutionLength; i++)
460                         {
461                             substitutionArray[i] = ulongSubstitutionArray[i];
462                         }
463                         ulong Callback(FfiLink_UInt64 before, FfiLink_UInt64 after) =>
                                (ulong)from_t.Convert(handler?.Invoke(new
                            ↪   Link<TLinkAddress>(from_u64.Convert(before.Index),
                            ↪   from_u64.Convert(before.Source), from_u64.Convert(before.Target)),
                            ↪   new Link<TLinkAddress>(from_u64.Convert(after.Index),
                            ↪   from_u64.Convert(after.Source), from_u64.Convert(after.Target))) ??
                            ↪   Constants.Continue);
464                         return from_u64.Convert(Methods.UInt64UnitedMemoryLinks_Update(_ptr,
                            ↪   restrictionArray, (nuint)restrictionLength, substitutionArray,
                            ↪   (nuint)(substitution?.Count ?? 0), Callback));
465                     }
466                     default:
467                     {
468                         throw new NotImplementedException();
469                     }
470                 }
471             }
472         }
473
474         public TLinkAddress Delete(IList<TLinkAddress>? restriction, WriteHandler<TLinkAddress>?
            ↪   handler)
475         {
476             var restrictionLength = restriction?.Count ?? 0;
477             unsafe
478             {
479                 TLinkAddress t = default;
480                 switch (t)
481                 {
482                     case byte:
483                     {
484                         var restrictionArray = stackalloc byte[restrictionLength];
485                         var byteRestrictionArray = (IList<byte>)restriction;
486                         for (var i = 0; i < restrictionLength; i++)
487                         {
488                             restrictionArray[i] = byteRestrictionArray[i];
489                         }
490                         byte Callback(FfiLink_UInt8 before, FfiLink_UInt8 after) =>
                                (byte)from_t.Convert(handler?.Invoke(new
                            ↪   Link<TLinkAddress>(from_u8.Convert(before.Index),
                            ↪   from_u8.Convert(before.Source), from_u8.Convert(before.Target)), new
                            ↪   Link<TLinkAddress>(from_u8.Convert(after.Index),
                            ↪   from_u8.Convert(after.Source), from_u8.Convert(after.Target))) ??
                            ↪   Constants.Continue);
491                         return (TLinkAddress)(object)Methods.ByteUnitedMemoryLinks_Delete(_ptr,
                            ↪   restrictionArray, (nuint)restrictionLength, Callback);
492                     }
493                     case ushort:
494                     {
495                         var restrictionArray = stackalloc ushort[restrictionLength];
496                         var ushortRestrictionArray = (IList<ushort>)restriction;
497                         for (var i = 0; i < restrictionLength; i++)
498                         {
499                             restrictionArray[i] = ushortRestrictionArray[i];
```

```
500                         }
501                         ushort Callback(FfiLink_UInt16 before, FfiLink_UInt16 after) =>
                                ↪ (ushort)from_t.Convert(handler?.Invoke(new
                                ↪ Link<TLinkAddress>(from_u16.Convert(before.Index),
                                ↪ from_u16.Convert(before.Source), from_u16.Convert(before.Target)),
                                ↪ new Link<TLinkAddress>(from_u16.Convert(after.Index),
                                ↪ from_u16.Convert(after.Source), from_u16.Convert(after.Target))) ??
                                ↪ Constants.Continue);
502                         return
                                ↪ (TLinkAddress)(object)Methods.UInt16UnitedMemoryLinks_Delete(_ptr,
                                ↪ restrictionArray, (nuint)restrictionLength, Callback);
503                     }
504                     case uint:
505                     {
506                         var restrictionArray = stackalloc uint[restrictionLength];
507                         var uintRestrictionArray = (IList<uint>)restriction;
508                         for (var i = 0; i < restrictionLength; i++)
509                         {
510                             restrictionArray[i] = uintRestrictionArray[i];
511                         }
512                         uint Callback(FfiLink_UInt32 before, FfiLink_UInt32 after) =>
                                ↪ (uint)from_t.Convert(handler?.Invoke(new
                                ↪ Link<TLinkAddress>(from_u32.Convert(before.Index),
                                ↪ from_u32.Convert(before.Source), from_u32.Convert(before.Target)),
                                ↪ new Link<TLinkAddress>(from_u32.Convert(after.Index),
                                ↪ from_u32.Convert(after.Source), from_u32.Convert(after.Target))) ??
                                ↪ Constants.Continue);
513                         return
                                ↪ (TLinkAddress)(object)Methods.UInt32UnitedMemoryLinks_Delete(_ptr,
                                ↪ restrictionArray, (nuint)restrictionLength, Callback);
514                     }
515                     case ulong:
516                     {
517                         var restrictionArray = stackalloc ulong[restrictionLength];
518                         var ulongRestrictionArray = (IList<ulong>)restriction;
519                         for (var i = 0; i < restrictionLength; i++)
520                         {
521                             restrictionArray[i] = ulongRestrictionArray[i];
522                         }
523                         ulong Callback(FfiLink_UInt64 before, FfiLink_UInt64 after) =>
                                ↪ (ulong)from_t.Convert(handler?.Invoke(new
                                ↪ Link<TLinkAddress>(from_u64.Convert(before.Index),
                                ↪ from_u64.Convert(before.Source), from_u64.Convert(before.Target)),
                                ↪ new Link<TLinkAddress>(from_u64.Convert(after.Index),
                                ↪ from_u64.Convert(after.Source), from_u64.Convert(after.Target))) ??
                                ↪ Constants.Continue);
524                         return
                                ↪ (TLinkAddress)(object)Methods.UInt64UnitedMemoryLinks_Delete(_ptr,
                                ↪ restrictionArray, (nuint)restrictionLength, Callback);
525                     }
526                     default:
527                     {
528                         throw new NotImplementedException();
529                     }
530                 }
531             }
532         }
533
534         protected override void Dispose(bool manual, bool wasDisposed)
535         {
536             unsafe
537             {
538                 if (wasDisposed && _ptr != null)
539                 {
540                     return;
541                 }
542                 TLinkAddress t = default;
543                 switch (t)
544                     {
545                         case byte:
546                             Methods.ByteUnitedMemoryLinks_Drop(_ptr);
547                             break;
548                         case ushort:
549                             Methods.UInt16UnitedMemoryLinks_Drop(_ptr);
550                             break;
551                         case uint:
552                             Methods.UInt32UnitedMemoryLinks_Drop(_ptr);
553                             break;
554                         case ulong:
```

```
                            Methods.UInt64UnitedMemoryLinks_Drop(_ptr);
                            break;
                        default:
                            throw new NotImplementedException();
                    }
                }
            }
        }
```

1.22 ./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
{
    /// <summary>
    /// <para>
    /// Defines the links.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ILinks{TLinkAddress, LinksConstants{TLinkAddress}}"/>
    public interface ILinks<TLinkAddress> : ILinks<TLinkAddress, LinksConstants<TLinkAddress>>
    {
    }
}
```

1.23 ./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.Lists;
using Platform.Random;
using Platform.Setters;
using Platform.Converters;
using Platform.Numbers;
using Platform.Data.Exceptions;
using Platform.Data.Doublets.Decorators;
using Platform.Delegates;

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

namespace Platform.Data.Doublets
{
    /// <summary>
    /// <para>
    /// Represents the links extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class ILinksExtensions
    {
        /// <summary>
        /// <para>
        /// Runs the random creations using the specified links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TLinkAddress">
        /// <para>The link.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="links">
        /// <para>The links.</para>
        /// <para></para>
        /// </param>
        /// <param name="amountOfCreations">
        /// <para>The amount of creations.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void RunRandomCreations<TLinkAddress>(this ILinks<TLinkAddress> links,
        ↪  ulong amountOfCreations)
        {
```

```
            var random = RandomHelpers.Default;
            var addressToUInt64Converter = UncheckedConverter<TLinkAddress, ulong>.Default;
            var uInt64ToAddressConverter = UncheckedConverter<ulong, TLinkAddress>.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);
            }
        }

        /// <summary>
        /// <para>
        /// Runs the random searches using the specified links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TLinkAddress">
        /// <para>The link.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="links">
        /// <para>The links.</para>
        /// <para></para>
        /// </param>
        /// <param name="amountOfSearches">
        /// <para>The amount of searches.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void RunRandomSearches<TLinkAddress>(this ILinks<TLinkAddress> links,
        ↪  ulong amountOfSearches)
        {
            var random = RandomHelpers.Default;
            var addressToUInt64Converter = UncheckedConverter<TLinkAddress, ulong>.Default;
            var uInt64ToAddressConverter = UncheckedConverter<ulong, TLinkAddress>.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);
            }
        }

        /// <summary>
        /// <para>
        /// Runs the random deletions using the specified links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TLinkAddress">
        /// <para>The link.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="links">
        /// <para>The links.</para>
        /// <para></para>
        /// </param>
        /// <param name="amountOfDeletions">
        /// <para>The amount of deletions.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void RunRandomDeletions<TLinkAddress>(this ILinks<TLinkAddress> links,
        ↪  ulong amountOfDeletions)
        {
            var random = RandomHelpers.Default;
            var addressToUInt64Converter = UncheckedConverter<TLinkAddress, ulong>.Default;
            var uInt64ToAddressConverter = UncheckedConverter<ulong, TLinkAddress>.Default;
            var linksCount = addressToUInt64Converter.Convert(links.Count());
```

```
            var min = amountOfDeletions > linksCount ? OUL : linksCount - amountOfDeletions;
            for (var i = OUL; 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);
            }
        }

        /// <summary>
        /// <para>
        /// Deletes the links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TLinkAddress">
        /// <para>The link.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="links">
        /// <para>The links.</para>
        /// <para></para>
        /// </param>
        /// <param name="linkToDelete">
        /// <para>The link to delete.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLinkAddress Delete<TLinkAddress>(this ILinks<TLinkAddress> links,
        ↪  TLinkAddress linkToDelete, WriteHandler<TLinkAddress>? handler)
        {
            if (links.Exists(linkToDelete))
            {
                links.EnforceResetValues(linkToDelete, handler);
            }
            return links.Delete(new LinkAddress<TLinkAddress>(linkToDelete), handler);
        }

        /// <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<TLinkAddress>(this ILinks<TLinkAddress> links)
        {
            var equalityComparer = EqualityComparer<TLinkAddress>.Default;
            var comparer = Comparer<TLinkAddress>.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();
                }
            }
        }

        /// <summary>
        /// <para>
        /// Firsts the links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TLinkAddress">
        /// <para>The link.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="links">
        /// <para>The links.</para>
```

```
        /// <para></para>
        /// </param>
        /// <exception cref="InvalidOperationException">
        /// <para>В процессе поиска по хранилищу не было найдено связей.</para>
        /// <para></para>
        /// </exception>
        /// <exception cref="InvalidOperationException">
        /// <para>В хранилище нет связей.</para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The first link.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLinkAddress First<TLinkAddress>(this ILinks<TLinkAddress> links)
        {
            TLinkAddress firstLink = default;
            var equalityComparer = EqualityComparer<TLinkAddress>.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;
        }

        /// <summary>
        /// <para>
        /// Singles the or default using the specified links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TLinkAddress">
        /// <para>The link.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="links">
        /// <para>The links.</para>
        /// <para></para>
        /// </param>
        /// <param name="query">
        /// <para>The query.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The result.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IList<TLinkAddress>? SingleOrDefault<TLinkAddress>(this
        ↪  ILinks<TLinkAddress> links, IList<TLinkAddress>? query)
        {
            IList<TLinkAddress>? result = null;
            var count = 0;
            var constants = links.Constants;
            var @continue = constants.Continue;
            var @break = constants.Break;
            links.Each(query, linkHandler);
            return result;

            TLinkAddress linkHandler(IList<TLinkAddress>? link)
            {
                if (count == 0)
                {
                    result = link;
                    count++;
                    return @continue;
                }
                else
```

```
271                 {
272                     result = null;
273                     return @break;
274                 }
275             }
276         }
277 
278         #region Paths
279 
280         /// <remarks>
281         /// TODO: Как так? Как то что ниже может быть корректно?
282         /// Скорее всего практически не применимо
283         /// Предполагалось, что можно было конвертировать формируемый в проходе через
            ↪ SequenceWalker
284         /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
285         /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
286         /// </remarks>
287         [MethodImpl(MethodImplOptions.AggressiveInlining)]
288         public static bool CheckPathExistance<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪ params TLinkAddress[] path)
289         {
290             var current = path[0];
291             //EnsureLinkExists(current, "path");
292             if (!links.Exists(current))
293             {
294                 return false;
295             }
296             var equalityComparer = EqualityComparer<TLinkAddress>.Default;
297             var constants = links.Constants;
298             for (var i = 1; i < path.Length; i++)
299             {
300                 var next = path[i];
301                 var values = links.GetLink(current);
302                 var source = values[constants.SourcePart];
303                 var target = values[constants.TargetPart];
304                 if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
                    ↪ next))
305                 {
306                     //throw new InvalidOperationException(string.Format("Невозможно выбрать
                        ↪ путь, так как и Source и Target совпадают с элементом пути {0}.", next));
307                     return false;
308                 }
309                 if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
                    ↪ target))
310                 {
311                     //throw new InvalidOperationException(string.Format("Невозможно продолжить
                        ↪ путь через элемент пути {0}", next));
312                     return false;
313                 }
314                 current = next;
315             }
316             return true;
317         }
318 
319         /// <remarks>
320         /// Может потребовать дополнительного стека для PathElement's при использовании
            ↪ SequenceWalker.
321         /// </remarks>
322         [MethodImpl(MethodImplOptions.AggressiveInlining)]
323         public static TLinkAddress GetByKeys<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪ TLinkAddress root, params int[] path)
324         {
325             links.EnsureLinkExists(root, "root");
326             var currentLink = root;
327             for (var i = 0; i < path.Length; i++)
328             {
329                 currentLink = links.GetLink(currentLink)[path[i]];
330             }
331             return currentLink;
332         }
333 
334         /// <summary>
335         /// <para>
336         /// Gets the square matrix sequence element by index using the specified links.
337         /// </para>
338         /// <para></para>
339         /// </summary>
340         /// <typeparam name="TLinkAddress">
```

```
341         /// <para>The link.</para>
342         /// <para></para>
343         /// </typeparam>
344         /// <param name="links">
345         /// <para>The links.</para>
346         /// <para></para>
347         /// </param>
348         /// <param name="root">
349         /// <para>The root.</para>
350         /// <para></para>
351         /// </param>
352         /// <param name="size">
353         /// <para>The size.</para>
354         /// <para></para>
355         /// </param>
356         /// <param name="index">
357         /// <para>The index.</para>
358         /// <para></para>
359         /// </param>
360         /// <exception cref="ArgumentOutOfRangeException">
361         /// <para>Sequences with sizes other than powers of two are not supported.</para>
362         /// <para></para>
363         /// </exception>
364         /// <returns>
365         /// <para>The current link.</para>
366         /// <para></para>
367         /// </returns>
368         [MethodImpl(MethodImplOptions.AggressiveInlining)]
369         public static TLinkAddress GetSquareMatrixSequenceElementByIndex<TLinkAddress>(this
              ↪ ILinks<TLinkAddress> links, TLinkAddress root, ulong size, ulong index)
370         {
371             var constants = links.Constants;
372             var source = constants.SourcePart;
373             var target = constants.TargetPart;
374             if (!Platform.Numbers.Math.IsPowerOfTwo(size))
375             {
376                 throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes other
                      ↪  than powers of two are not supported.");
377             }
378             var path = new BitArray(BitConverter.GetBytes(index));
379             var length = Bit.GetLowestPosition(size);
380             links.EnsureLinkExists(root, "root");
381             var currentLink = root;
382             for (var i = length - 1; i >= 0; i--)
383             {
384                 currentLink = links.GetLink(currentLink)[path[i] ? target : source];
385             }
386             return currentLink;
387         }
388 
389         #endregion
390 
391         /// <summary>
392         /// Возвращает индекс указанной связи.
393         /// </summary>
394         /// <param name="links">Хранилище связей.</param>
395         /// <param name="link">Связь представленная списком, состоящим из её адреса и
              ↪ содержимого.</param>
396         /// <returns>Индекс начальной связи для указанной связи.</returns>
397         [MethodImpl(MethodImplOptions.AggressiveInlining)]
398         public static TLinkAddress GetIndex<TLinkAddress>(this ILinks<TLinkAddress> links,
              ↪ IList<TLinkAddress>? link) => link[links.Constants.IndexPart];
399 
400         /// <summary>
401         /// Возвращает индекс начальной (Source) связи для указанной связи.
402         /// </summary>
403         /// <param name="links">Хранилище связей.</param>
404         /// <param name="link">Индекс связи.</param>
405         /// <returns>Индекс начальной связи для указанной связи.</returns>
406         [MethodImpl(MethodImplOptions.AggressiveInlining)]
407         public static TLinkAddress GetSource<TLinkAddress>(this ILinks<TLinkAddress> links,
              ↪ TLinkAddress link) => links.GetLink(link)[links.Constants.SourcePart];
408 
409         /// <summary>
410         /// Возвращает индекс начальной (Source) связи для указанной связи.
411         /// </summary>
412         /// <param name="links">Хранилище связей.</param>
```

```
413         /// <param name="link">Связь представленная списком, состоящим из её адреса и
            ↪  содержимого.</param>
414         /// <returns>Индекс начальной связи для указанной связи.</returns>
415         [MethodImpl(MethodImplOptions.AggressiveInlining)]
416         public static TLinkAddress GetSource<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪  IList<TLinkAddress>? link) => link[links.Constants.SourcePart];
417
418         /// <summary>
419         /// Возвращает индекс конечной (Target) связи для указанной связи.
420         /// </summary>
421         /// <param name="links">Хранилище связей.</param>
422         /// <param name="link">Индекс связи.</param>
423         /// <returns>Индекс конечной связи для указанной связи.</returns>
424         [MethodImpl(MethodImplOptions.AggressiveInlining)]
425         public static TLinkAddress GetTarget<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪  TLinkAddress link) => links.GetLink(link)[links.Constants.TargetPart];
426
427         /// <summary>
428         /// Возвращает индекс конечной (Target) связи для указанной связи.
429         /// </summary>
430         /// <param name="links">Хранилище связей.</param>
431         /// <param name="link">Связь представленная списком, состоящим из её адреса и
            ↪  содержимого.</param>
432         /// <returns>Индекс конечной связи для указанной связи.</returns>
433         [MethodImpl(MethodImplOptions.AggressiveInlining)]
434         public static TLinkAddress GetTarget<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪  IList<TLinkAddress>? link) => link[links.Constants.TargetPart];
435
436         /// <summary>
437         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪  (handler) для каждой подходящей связи.
438         /// </summary>
439         /// <param name="links">Хранилище связей.</param>
440         /// <param name="handler">Обработчик каждой подходящей связи.</param>
441         /// <param name="restriction">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>
442         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪  случае.</returns>
443         [MethodImpl(MethodImplOptions.AggressiveInlining)]
444         public static bool Each<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪  ReadHandler<TLinkAddress>? handler, params TLinkAddress[] restriction)
445             => EqualityComparer<TLinkAddress>.Default.Equals(links.Each(restriction, handler),
                ↪  links.Constants.Continue);
446
447         public static bool Each<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪  Func<TLinkAddress, bool> handler, TLinkAddress source, TLinkAddress target) =>
            ↪  links.Each(source, target, handler);
448
449
450         /// <summary>
451         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪  (handler) для каждой подходящей связи.
452         /// </summary>
453         /// <param name="links">Хранилище связей.</param>
454         /// <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>
455         /// <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>
456         /// <param name="handler">Обработчик каждой подходящей связи.</param>
457         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪  случае.</returns>
458         [MethodImpl(MethodImplOptions.AggressiveInlining)]
459         public static bool Each<TLinkAddress>(this ILinks<TLinkAddress> links, TLinkAddress
            ↪  source, TLinkAddress target, Func<TLinkAddress, bool> handler)
460         {
461             var constants = links.Constants;
462             return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
                ↪  constants.Break, constants.Any, source, target);
463         }
464
465         public static bool Each<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪  ReadHandler<TLinkAddress>? handler, TLinkAddress source, TLinkAddress target) =>
            ↪  links.Each(source, target, handler);
```

```
466
467
468         /// <summary>
469         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
                ↪ (handler) для каждой подходящей связи.
470         /// </summary>
471         /// <param name="links">Хранилище связей.</param>
472         /// <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>
473         /// <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>
474         /// <param name="handler">Обработчик каждой подходящей связи.</param>
475         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
                ↪ случае.</returns>
476         [MethodImpl(MethodImplOptions.AggressiveInlining)]
477         public static bool Each<TLinkAddress>(this ILinks<TLinkAddress> links, TLinkAddress
                ↪ source, TLinkAddress target, ReadHandler<TLinkAddress>? handler) =>
                ↪ links.Each(handler, links.Constants.Any, source, target);
478
479         /// <summary>
480         /// <para>
481         /// Alls the links.
482         /// </para>
483         /// <para></para>
484         /// </summary>
485         /// <typeparam name="TLinkAddress">
486         /// <para>The link.</para>
487         /// <para></para>
488         /// </typeparam>
489         /// <param name="links">
490         /// <para>The links.</para>
491         /// <para></para>
492         /// </param>
493         /// <param name="restriction">
494         /// <para>The restriction.</para>
495         /// <para></para>
496         /// </param>
497         /// <returns>
498         /// <para>A list of i list t link</para>
499         /// <para></para>
500         /// </returns>
501         [MethodImpl(MethodImplOptions.AggressiveInlining)]
502         public static IList<IList<TLinkAddress>?> All<TLinkAddress>(this ILinks<TLinkAddress>
                ↪ links, params TLinkAddress[] restriction)
503         {
504             var allLinks = new List<IList<TLinkAddress>?>();
505             var filler = new ListFiller<IList<TLinkAddress>?, TLinkAddress>(allLinks,
                    ↪ links.Constants.Continue);
506             links.Each(filler.AddAndReturnConstant, restriction);
507             return allLinks;
508         }
509
510         /// <summary>
511         /// <para>
512         /// Alls the indices using the specified links.
513         /// </para>
514         /// <para></para>
515         /// </summary>
516         /// <typeparam name="TLinkAddress">
517         /// <para>The link.</para>
518         /// <para></para>
519         /// </typeparam>
520         /// <param name="links">
521         /// <para>The links.</para>
522         /// <para></para>
523         /// </param>
524         /// <param name="restriction">
525         /// <para>The restriction.</para>
526         /// <para></para>
527         /// </param>
528         /// <returns>
529         /// <para>A list of t link</para>
530         /// <para></para>
531         /// </returns>
532         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
533         public static IList<TLinkAddress>? AllIndices<TLinkAddress>(this ILinks<TLinkAddress>
            ↪ links, params TLinkAddress[] restriction)
534         {
535             var allIndices = new List<TLinkAddress>();
536             var filler = new ListFiller<TLinkAddress, TLinkAddress>(allIndices,
                ↪ links.Constants.Continue);
537             links.Each(filler.AddFirstAndReturnConstant, restriction);
538             return allIndices;
539         }
540 
541         /// <summary>
542         /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
            ↪ в хранилище связей.
543         /// </summary>
544         /// <param name="links">Хранилище связей.</param>
545         /// <param name="source">Начало связи.</param>
546         /// <param name="target">Конец связи.</param>
547         /// <returns>Значение, определяющее существует ли связь.</returns>
548         [MethodImpl(MethodImplOptions.AggressiveInlining)]
549         public static bool Exists<TLinkAddress>(this ILinks<TLinkAddress> links, TLinkAddress
            ↪ source, TLinkAddress target) =>
            ↪ Comparer<TLinkAddress>.Default.Compare(links.Count(links.Constants.Any, source,
            ↪ target), default) > 0;
550 
551         #region Ensure
552         // TODO: May be move to EnsureExtensions or make it both there and here
553 
554         /// <summary>
555         /// <para>
556         /// Ensures the link exists using the specified links.
557         /// </para>
558         /// <para></para>
559         /// </summary>
560         /// <typeparam name="TLinkAddress">
561         /// <para>The link.</para>
562         /// <para></para>
563         /// </typeparam>
564         /// <param name="links">
565         /// <para>The links.</para>
566         /// <para></para>
567         /// </param>
568         /// <param name="restriction">
569         /// <para>The restriction.</para>
570         /// <para></para>
571         /// </param>
572         /// <exception cref="ArgumentLinkDoesNotExistsException{TLinkAddress}">
573         /// <para>sequence[{i}]</para>
574         /// <para></para>
575         /// </exception>
576         [MethodImpl(MethodImplOptions.AggressiveInlining)]
577         public static void EnsureLinkExists<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪ IList<TLinkAddress>? restriction)
578         {
579             for (var i = 0; i < restriction.Count; i++)
580             {
581                 if (!links.Exists(restriction[i]))
582                 {
583                     throw new ArgumentLinkDoesNotExistsException<TLinkAddress>(restriction[i],
                        ↪ $"sequence[{i}]");
584                 }
585             }
586         }
587 
588         /// <summary>
589         /// <para>
590         /// Ensures the inner reference exists using the specified links.
591         /// </para>
592         /// <para></para>
593         /// </summary>
594         /// <typeparam name="TLinkAddress">
595         /// <para>The link.</para>
596         /// <para></para>
597         /// </typeparam>
598         /// <param name="links">
599         /// <para>The links.</para>
600         /// <para></para>
601         /// </param>
```

```
602         /// <param name="reference">
603         /// <para>The reference.</para>
604         /// <para></para>
605         /// </param>
606         /// <param name="argumentName">
607         /// <para>The argument name.</para>
608         /// <para></para>
609         /// </param>
610         /// <exception cref="ArgumentLinkDoesNotExistsException{TLinkAddress}">
611         /// <para></para>
612         /// <para></para>
613         /// </exception>
614         [MethodImpl(MethodImplOptions.AggressiveInlining)]
615         public static void EnsureInnerReferenceExists<TLinkAddress>(this ILinks<TLinkAddress>
            ↪ links, TLinkAddress reference, string argumentName)
616         {
617             if (links.Constants.IsInternalReference(reference) && !links.Exists(reference))
618             {
619                 throw new ArgumentLinkDoesNotExistsException<TLinkAddress>(reference,
                    ↪ argumentName);
620             }
621         }
622
623         /// <summary>
624         /// <para>
625         /// Ensures the inner reference exists using the specified links.
626         /// </para>
627         /// <para></para>
628         /// </summary>
629         /// <typeparam name="TLinkAddress">
630         /// <para>The link.</para>
631         /// <para></para>
632         /// </typeparam>
633         /// <param name="links">
634         /// <para>The links.</para>
635         /// <para></para>
636         /// </param>
637         /// <param name="restriction">
638         /// <para>The restriction.</para>
639         /// <para></para>
640         /// </param>
641         /// <param name="argumentName">
642         /// <para>The argument name.</para>
643         /// <para></para>
644         /// </param>
645         [MethodImpl(MethodImplOptions.AggressiveInlining)]
646         public static void EnsureInnerReferenceExists<TLinkAddress>(this ILinks<TLinkAddress>
            ↪ links, IList<TLinkAddress>? restriction, string argumentName)
647         {
648             for (int i = 0; i < restriction.Count; i++)
649             {
650                 links.EnsureInnerReferenceExists(restriction[i], argumentName);
651             }
652         }
653
654         /// <summary>
655         /// <para>
656         /// Ensures the link is any or exists using the specified links.
657         /// </para>
658         /// <para></para>
659         /// </summary>
660         /// <typeparam name="TLinkAddress">
661         /// <para>The link.</para>
662         /// <para></para>
663         /// </typeparam>
664         /// <param name="links">
665         /// <para>The links.</para>
666         /// <para></para>
667         /// </param>
668         /// <param name="restriction">
669         /// <para>The restriction.</para>
670         /// <para></para>
671         /// </param>
672         /// <exception cref="ArgumentLinkDoesNotExistsException{TLinkAddress}">
673         /// <para>sequence[{i}]</para>
674         /// <para></para>
675         /// </exception>
```

```
676         [MethodImpl(MethodImplOptions.AggressiveInlining)]
677         public static void EnsureLinkIsAnyOrExists<TLinkAddress>(this ILinks<TLinkAddress>
            ↪ links, IList<TLinkAddress>? restriction)
678         {
679             var equalityComparer = EqualityComparer<TLinkAddress>.Default;
680             var any = links.Constants.Any;
681             for (var i = 0; i < restriction.Count; i++)
682             {
683                 if (!equalityComparer.Equals(restriction[i], any) &&
                    ↪ !links.Exists(restriction[i]))
684                 {
685                     throw new ArgumentLinkDoesNotExistsException<TLinkAddress>(restriction[i],
                        ↪ $"sequence[{i}]");
686                 }
687             }
688         }
689
690         /// <summary>
691         /// <para>
692         /// Ensures the link is any or exists using the specified links.
693         /// </para>
694         /// <para></para>
695         /// </summary>
696         /// <typeparam name="TLinkAddress">
697         /// <para>The link.</para>
698         /// <para></para>
699         /// </typeparam>
700         /// <param name="links">
701         /// <para>The links.</para>
702         /// <para></para>
703         /// </param>
704         /// <param name="link">
705         /// <para>The link.</para>
706         /// <para></para>
707         /// </param>
708         /// <param name="argumentName">
709         /// <para>The argument name.</para>
710         /// <para></para>
711         /// </param>
712         /// <exception cref="ArgumentLinkDoesNotExistsException{TLinkAddress}">
713         /// <para></para>
714         /// <para></para>
715         /// </exception>
716         [MethodImpl(MethodImplOptions.AggressiveInlining)]
717         public static void EnsureLinkIsAnyOrExists<TLinkAddress>(this ILinks<TLinkAddress>
            ↪ links, TLinkAddress link, string argumentName)
718         {
719             var equalityComparer = EqualityComparer<TLinkAddress>.Default;
720             if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
721             {
722                 throw new ArgumentLinkDoesNotExistsException<TLinkAddress>(link, argumentName);
723             }
724         }
725
726         /// <summary>
727         /// <para>
728         /// Ensures the link is itself or exists using the specified links.
729         /// </para>
730         /// <para></para>
731         /// </summary>
732         /// <typeparam name="TLinkAddress">
733         /// <para>The link.</para>
734         /// <para></para>
735         /// </typeparam>
736         /// <param name="links">
737         /// <para>The links.</para>
738         /// <para></para>
739         /// </param>
740         /// <param name="link">
741         /// <para>The link.</para>
742         /// <para></para>
743         /// </param>
744         /// <param name="argumentName">
745         /// <para>The argument name.</para>
746         /// <para></para>
747         /// </param>
748         /// <exception cref="ArgumentLinkDoesNotExistsException{TLinkAddress}">
749         /// <para></para>
```

```
750             /// <para></para>
751             /// </exception>
752             [MethodImpl(MethodImplOptions.AggressiveInlining)]
753             public static void EnsureLinkIsItselfOrExists<TLinkAddress>(this ILinks<TLinkAddress>
                ↪  links, TLinkAddress link, string argumentName)
754             {
755                 var equalityComparer = EqualityComparer<TLinkAddress>.Default;
756                 if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
757                 {
758                     throw new ArgumentLinkDoesNotExistsException<TLinkAddress>(link, argumentName);
759                 }
760             }
761
762             /// <param name="links">Хранилище связей.</param>
763             [MethodImpl(MethodImplOptions.AggressiveInlining)]
764             public static void EnsureDoesNotExists<TLinkAddress>(this ILinks<TLinkAddress> links,
                ↪  TLinkAddress source, TLinkAddress target)
765             {
766                 if (links.Exists(source, target))
767                 {
768                     throw new LinkWithSameValueAlreadyExistsException();
769                 }
770             }
771
772             /// <param name="links">Хранилище связей.</param>
773             [MethodImpl(MethodImplOptions.AggressiveInlining)]
774             public static void EnsureNoUsages<TLinkAddress>(this ILinks<TLinkAddress> links,
                ↪  TLinkAddress link)
775             {
776                 if (links.HasUsages(link))
777                 {
778                     throw new ArgumentLinkHasDependenciesException<TLinkAddress>(link);
779                 }
780             }
781
782             /// <param name="links">Хранилище связей.</param>
783             [MethodImpl(MethodImplOptions.AggressiveInlining)]
784             public static void EnsureCreated<TLinkAddress>(this ILinks<TLinkAddress> links, params
                ↪  TLinkAddress[] addresses) => links.EnsureCreated(links.Create, addresses);
785
786             /// <param name="links">Хранилище связей.</param>
787             [MethodImpl(MethodImplOptions.AggressiveInlining)]
788             public static void EnsurePointsCreated<TLinkAddress>(this ILinks<TLinkAddress> links,
                ↪  params TLinkAddress[] addresses) => links.EnsureCreated(links.CreatePoint,
                ↪  addresses);
789
790             /// <param name="links">Хранилище связей.</param>
791             [MethodImpl(MethodImplOptions.AggressiveInlining)]
792             public static void EnsureCreated<TLinkAddress>(this ILinks<TLinkAddress> links,
                ↪  Func<TLinkAddress> creator, params TLinkAddress[] addresses)
793             {
794                 var addressToUInt64Converter = CheckedConverter<TLinkAddress, ulong>.Default;
795                 var uInt64ToAddressConverter = CheckedConverter<ulong, TLinkAddress>.Default;
796                 var nonExistentAddresses = new HashSet<TLinkAddress>(addresses.Where(x =>
                    ↪  !links.Exists(x)));
797                 if (nonExistentAddresses.Count > 0)
798                 {
799                     var max = nonExistentAddresses.Max();
800                     max = uInt64ToAddressConverter.Convert(System.Math.Min(addressToUInt64Converter.
                        ↪  Convert(max),
                        ↪  addressToUInt64Converter.Convert(links.Constants.InternalReferencesRange.Max
                        ↪  imum)));
801                     var createdLinks = new List<TLinkAddress>();
802                     var equalityComparer = EqualityComparer<TLinkAddress>.Default;
803                     TLinkAddress createdLink = creator();
804                     while (!equalityComparer.Equals(createdLink, max))
805                     {
806                         createdLinks.Add(createdLink);
807                     }
808                     for (var i = 0; i < createdLinks.Count; i++)
809                     {
810                         if (!nonExistentAddresses.Contains(createdLinks[i]))
811                         {
812                             links.Delete(createdLinks[i]);
813                         }
814                     }
815                 }
```

```
816         }
817
818         #endregion
819
820         /// <param name="links">Хранилище связей.</param>
821         [MethodImpl(MethodImplOptions.AggressiveInlining)]
822         public static TLinkAddress CountUsages<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪ TLinkAddress link)
823         {
824             var constants = links.Constants;
825             var values = links.GetLink(link);
826             TLinkAddress usagesAsSource = links.Count(new Link<TLinkAddress>(constants.Any,
                ↪ link, constants.Any));
827             var equalityComparer = EqualityComparer<TLinkAddress>.Default;
828             if (equalityComparer.Equals(values[constants.SourcePart], link))
829             {
830                 usagesAsSource = Arithmetic<TLinkAddress>.Decrement(usagesAsSource);
831             }
832             TLinkAddress usagesAsTarget = links.Count(new Link<TLinkAddress>(constants.Any,
                ↪ constants.Any, link));
833             if (equalityComparer.Equals(values[constants.TargetPart], link))
834             {
835                 usagesAsTarget = Arithmetic<TLinkAddress>.Decrement(usagesAsTarget);
836             }
837             return Arithmetic<TLinkAddress>.Add(usagesAsSource, usagesAsTarget);
838         }
839
840         /// <param name="links">Хранилище связей.</param>
841         [MethodImpl(MethodImplOptions.AggressiveInlining)]
842         public static bool HasUsages<TLinkAddress>(this ILinks<TLinkAddress> links, TLinkAddress
            ↪ link) => Comparer<TLinkAddress>.Default.Compare(links.CountUsages(link), default) >
            ↪ 0;
843
844         /// <param name="links">Хранилище связей.</param>
845         [MethodImpl(MethodImplOptions.AggressiveInlining)]
846         public static bool Equals<TLinkAddress>(this ILinks<TLinkAddress> links, TLinkAddress
            ↪ link, TLinkAddress source, TLinkAddress target)
847         {
848             var constants = links.Constants;
849             var values = links.GetLink(link);
850             var equalityComparer = EqualityComparer<TLinkAddress>.Default;
851             return equalityComparer.Equals(values[constants.SourcePart], source) &&
                ↪ equalityComparer.Equals(values[constants.TargetPart], target);
852         }
853
854         /// <summary>
855         /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
856         /// </summary>
857         /// <param name="links">Хранилище связей.</param>
858         /// <param name="source">Индекс связи, которая является началом для искомой
            ↪ связи.</param>
859         /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
860         /// <returns>Индекс искомой связи с указанными Source (началом) и Target
            ↪ (концом).</returns>
861         [MethodImpl(MethodImplOptions.AggressiveInlining)]
862         public static TLinkAddress SearchOrDefault<TLinkAddress>(this ILinks<TLinkAddress>
            ↪ links, TLinkAddress source, TLinkAddress target)
863         {
864             var contants = links.Constants;
865             var setter = new Setter<TLinkAddress, TLinkAddress>(contants.Continue,
                ↪ contants.Break, default);
866             links.Each(setter.SetFirstAndReturnFalse, contants.Any, source, target);
867             return setter.Result;
868         }
869
870         public static TLinkAddress CreatePoint<TLinkAddress>(this ILinks<TLinkAddress> links)
871         {
872             var constants = links.Constants;
873             var setter = new Setter<TLinkAddress, TLinkAddress>(constants.Continue,
                ↪ constants.Break);
874             links.CreatePoint(setter.SetFirstFromSecondListAndReturnTrue);
875             return setter.Result;
876         }
877
878         /// <param name="links">Хранилище связей.</param>
879         [MethodImpl(MethodImplOptions.AggressiveInlining)]
880         public static TLinkAddress CreatePoint<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪ WriteHandler<TLinkAddress>? handler)
```

```
881             {
882                 var constants = links.Constants;
883                 WriteHandlerState<TLinkAddress> handlerState = new(constants.Continue,
                    ↪  constants.Break, handler);
884                 TLinkAddress link = default;
885                 TLinkAddress HandlerWrapper(IList<TLinkAddress>? before, IList<TLinkAddress>? after)
886                 {
887                     link = after[constants.IndexPart];
888                     return handlerState.Handler != null ? handlerState.Handler(before, after) :
                        ↪  constants.Continue;
889                 }
890                 handlerState.Apply(links.Create(null, HandlerWrapper));
891                 handlerState.Apply(links.Update(link, link, link, HandlerWrapper));
892                 return handlerState.Result;
893             }
894 
895             public static TLinkAddress CreateAndUpdate<TLinkAddress>(this ILinks<TLinkAddress>
                ↪  links, TLinkAddress source, TLinkAddress target)
896             {
897                 var constants = links.Constants;
898                 var setter = new Setter<TLinkAddress, TLinkAddress>(constants.Continue,
                    ↪  constants.Break);
899                 links.CreateAndUpdate(source, target, setter.SetFirstFromSecondListAndReturnTrue);
900                 return setter.Result;
901             }
902 
903 
904             /// <param name="links">Хранилище связей.</param>
905             [MethodImpl(MethodImplOptions.AggressiveInlining)]
906             public static TLinkAddress CreateAndUpdate<TLinkAddress>(this ILinks<TLinkAddress>
                ↪  links, TLinkAddress source, TLinkAddress target, WriteHandler<TLinkAddress>? handler)
907             {
908                 var constants = links.Constants;
909                 TLinkAddress createdLink = default;
910                 WriteHandlerState<TLinkAddress> handlerState = new(constants.Continue,
                    ↪  constants.Break, handler);
911                 handlerState.Apply(links.Create(null, (before, after) =>
912                 {
913                     createdLink = links.GetIndex(after);
914                     return handlerState.Handler != null ? handlerState.Handler(before, after) :
                        ↪  constants.Continue;
915                 }));
916                 handlerState.Apply(links.Update(createdLink, source, target, handler));
917                 return handlerState.Result;
918             }
919 
920             /// <summary>
921             /// Обновляет связь с указанными началом (Source) и концом (Target)
922             /// на связь с указанными началом (NewSource) и концом (NewTarget).
923             /// </summary>
924             /// <param name="links">Хранилище связей.</param>
925             /// <param name="link">Индекс обновляемой связи.</param>
926             /// <param name="newSource">Индекс связи, которая является началом связи, на которую
                ↪  выполняется обновление.</param>
927             /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
                ↪  выполняется обновление.</param>
928             /// <returns>Индекс обновлённой связи.</returns>
929             [MethodImpl(MethodImplOptions.AggressiveInlining)]
930             public static TLinkAddress Update<TLinkAddress>(this ILinks<TLinkAddress> links,
                ↪  TLinkAddress link, TLinkAddress newSource, TLinkAddress newTarget) =>
                ↪  links.Update(new LinkAddress<TLinkAddress>(link), new Link<TLinkAddress>(link,
                ↪  newSource, newTarget));
931 
932             public static TLinkAddress Update<TLinkAddress>(this ILinks<TLinkAddress> links, params
                ↪  TLinkAddress[] restriction) => links.Update(restriction, null);
933 
934             public static TLinkAddress Update<TLinkAddress>(this ILinks<TLinkAddress> links,
                ↪  WriteHandler<TLinkAddress>? handler, params TLinkAddress[] restriction) =>
                ↪  links.Update(restriction, handler);
935 
936             public static TLinkAddress Update<TLinkAddress>(this ILinks<TLinkAddress> links,
                ↪  IList<TLinkAddress>? restriction)
937             {
938                 var constants = links.Constants;
939                 var setter = new Setter<TLinkAddress, TLinkAddress>(constants.Continue,
                    ↪  constants.Break);
940                 links.Update(restriction, setter.SetFirstFromSecondListAndReturnTrue);
941                 return setter.Result;
```

```
942         }
943
944
945         /// <summary>
946         /// Обновляет связь с указанными началом (Source) и концом (Target)
947         /// на связь с указанными началом (NewSource) и концом (NewTarget).
948         /// </summary>
949         /// <param name="links">Хранилище связей.</param>
950         /// <param name="restriction">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>
951         /// <returns>Индекс обновлённой связи.</returns>
952         [MethodImpl(MethodImplOptions.AggressiveInlining)]
953         public static TLinkAddress Update<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪ IList<TLinkAddress>? restriction, WriteHandler<TLinkAddress>? handler)
954         {
955             return restriction.Count switch
956             {
957                 2 => links.MergeAndDelete(restriction[0], restriction[1], handler),
958                 4 => links.UpdateOrCreateOrGet(restriction[0], restriction[1], restriction[2],
                    ↪  restriction[3], handler),
959                 _ => links.Update(restriction[0], restriction[1], restriction[2], handler)
960             };
961         }
962
963         public static TLinkAddress Update<TLinkAddress>(this ILinks<TLinkAddress> links,
            TLinkAddress link, TLinkAddress newSource, TLinkAddress newTarget,
            ↪ WriteHandler<TLinkAddress>? handler) => links.Update(new
            ↪ LinkAddress<TLinkAddress>(link), new Link<TLinkAddress>(link, newSource, newTarget),
            ↪ handler);
964
965         /// <summary>
966         /// <para>
967         /// Resolves the constant as self reference using the specified links.
968         /// </para>
969         /// <para></para>
970         /// </summary>
971         /// <typeparam name="TLinkAddress">
972         /// <para>The link.</para>
973         /// <para></para>
974         /// </typeparam>
975         /// <param name="links">
976         /// <para>The links.</para>
977         /// <para></para>
978         /// </param>
979         /// <param name="constant">
980         /// <para>The constant.</para>
981         /// <para></para>
982         /// </param>
983         /// <param name="restriction">
984         /// <para>The restriction.</para>
985         /// <para></para>
986         /// </param>
987         /// <param name="substitution">
988         /// <para>The substitution.</para>
989         /// <para></para>
990         /// </param>
991         /// <returns>
992         /// <para>A list of t link</para>
993         /// <para></para>
994         /// </returns>
995         [MethodImpl(MethodImplOptions.AggressiveInlining)]
996         public static IList<TLinkAddress>? ResolveConstantAsSelfReference<TLinkAddress>(this
            ↪ ILinks<TLinkAddress> links, TLinkAddress constant, IList<TLinkAddress>? restriction,
            ↪ IList<TLinkAddress>? substitution)
997         {
998             var equalityComparer = EqualityComparer<TLinkAddress>.Default;
999             var constants = links.Constants;
1000            var restrictionIndex = restriction[constants.IndexPart];
1001            var substitutionIndex = substitution[constants.IndexPart];
1002            if (equalityComparer.Equals(substitutionIndex, default))
1003            {
1004                substitutionIndex = restrictionIndex;
1005            }
1006            var source = substitution[constants.SourcePart];
1007            var target = substitution[constants.TargetPart];
1008            source = equalityComparer.Equals(source, constant) ? substitutionIndex : source;
1009            target = equalityComparer.Equals(target, constant) ? substitutionIndex : target;
```

```
            return new Link<TLinkAddress>(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 TLinkAddress GetOrCreate<TLinkAddress>(this ILinks<TLinkAddress> links,
        ↪  TLinkAddress source, TLinkAddress target)
        {
            var link = links.SearchOrDefault(source, target);
            if (EqualityComparer<TLinkAddress>.Default.Equals(link, default))
            {
                link = links.CreateAndUpdate(source, target);
            }
            return link;
        }

        public static TLinkAddress UpdateOrCreateOrGet<TLinkAddress>(this ILinks<TLinkAddress>
        ↪  links, TLinkAddress source, TLinkAddress target, TLinkAddress newSource,
        ↪  TLinkAddress newTarget)
        {
            var constants = links.Constants;
            var setter = new Setter<TLinkAddress, TLinkAddress>(constants.Continue,
            ↪  constants.Break);
            links.UpdateOrCreateOrGet(source, target, newSource, newTarget,
            ↪  setter.SetFirstFromSecondListAndReturnTrue);
            return setter.Result;
        }

        /// <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 TLinkAddress UpdateOrCreateOrGet<TLinkAddress>(this ILinks<TLinkAddress>
        ↪  links, TLinkAddress source, TLinkAddress target, TLinkAddress newSource,
        ↪  TLinkAddress newTarget, WriteHandler<TLinkAddress>? handler)
        {
            var equalityComparer = EqualityComparer<TLinkAddress>.Default;
            var link = links.SearchOrDefault(source, target);
            if (equalityComparer.Equals(link, default))
            {
                return links.CreateAndUpdate(newSource, newTarget, handler);
            }
            if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
            ↪  target))
            {
                var linkStruct = new Link<TLinkAddress>(link, source, target);
                return link;
            }
            return links.Update(link, newSource, newTarget, handler);
        }

        /// <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 TLinkAddress DeleteIfExists<TLinkAddress>(this ILinks<TLinkAddress> links,
        ↪  TLinkAddress source, TLinkAddress target)
```

```
        {
            var link = links.SearchOrDefault(source, target);
            if (!EqualityComparer<TLinkAddress>.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<TLinkAddress>(this ILinks<TLinkAddress> links,
        ↪  IList<TLinkAddress>? deletedLinks)
        {
            for (int i = 0; i < deletedLinks.Count; i++)
            {
                links.Delete(deletedLinks[i]);
            }
        }

        public static void DeleteAllUsages<TLinkAddress>(this ILinks<TLinkAddress> links,
        ↪  TLinkAddress linkIndex) => links.DeleteAllUsages(linkIndex, null);

        /// <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 TLinkAddress DeleteAllUsages<TLinkAddress>(this ILinks<TLinkAddress>
        ↪  links, TLinkAddress linkIndex, WriteHandler<TLinkAddress>? handler)
        {
            var constants = links.Constants;
            var any = constants.Any;
            var equalityComparer = EqualityComparer<TLinkAddress>.Default;
            var usagesAsSourceQuery = new Link<TLinkAddress>(any, linkIndex, any);
            var usagesAsTargetQuery = new Link<TLinkAddress>(any, any, linkIndex);
            var usages = new List<IList<TLinkAddress>?>();
            var usagesFiller = new ListFiller<IList<TLinkAddress>?, TLinkAddress>(usages,
            ↪  constants.Continue);
            links.Each(usagesFiller.AddAndReturnConstant, usagesAsSourceQuery);
            links.Each(usagesFiller.AddAndReturnConstant, usagesAsTargetQuery);
            WriteHandlerState<TLinkAddress> handlerState = new(constants.Continue,
            ↪  constants.Break, handler);
            foreach (var usage in usages)
            {
                if (equalityComparer.Equals(links.GetIndex(usage), linkIndex) ||
                ↪  !links.Exists(links.GetIndex(usage)))
                {
                    continue;
                }
                handlerState.Apply(links.Delete(links.GetIndex(usage), handlerState.Handler));
            }
            return handlerState.Result;
        }

        /// <summary>
        /// <para>
        /// Deletes the by query using the specified links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TLinkAddress">
        /// <para>The link.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="links">
        /// <para>The links.</para>
        /// <para></para>
        /// </param>
        /// <param name="query">
        /// <para>The query.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void DeleteByQuery<TLinkAddress>(this ILinks<TLinkAddress> links,
        ↪  Link<TLinkAddress> query)
```

```
        {
            var queryResult = new List<TLinkAddress>();
            var queryResultFiller = new ListFiller<TLinkAddress, TLinkAddress>(queryResult,
            ↪  links.Constants.Continue);
            links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
            foreach (var link in queryResult)
            {
                links.Delete(link);
            }
        }

        // TODO: Move to Platform.Data
        /// <summary>
        /// <para>
        /// Determines whether are values reset.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TLinkAddress">
        /// <para>The link.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="links">
        /// <para>The links.</para>
        /// <para></para>
        /// </param>
        /// <param name="linkIndex">
        /// <para>The link index.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool AreValuesReset<TLinkAddress>(this ILinks<TLinkAddress> links,
        ↪  TLinkAddress linkIndex)
        {
            var nullConstant = links.Constants.Null;
            var equalityComparer = EqualityComparer<TLinkAddress>.Default;
            var link = links.GetLink(linkIndex);
            for (int i = 1; i < link.Count; i++)
            {
                if (!equalityComparer.Equals(link[i], nullConstant))
                {
                    return false;
                }
            }
            return true;
        }

        public static void ResetValues<TLinkAddress>(this ILinks<TLinkAddress> links,
        ↪  TLinkAddress linkIndex) => links.ResetValues(linkIndex, null);

        // TODO: Create a universal version of this method in Platform.Data (with using of for
        ↪  loop)
        /// <summary>
        /// <para>
        /// Resets the values using the specified links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TLinkAddress">
        /// <para>The link.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="links">
        /// <para>The links.</para>
        /// <para></para>
        /// </param>
        /// <param name="linkIndex">
        /// <para>The link index.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLinkAddress ResetValues<TLinkAddress>(this ILinks<TLinkAddress> links,
        ↪  TLinkAddress linkIndex, WriteHandler<TLinkAddress>? handler)
        {
```

```
1214            var nullConstant = links.Constants.Null;
1215            var updateRequest = new Link<TLinkAddress>(linkIndex, nullConstant, nullConstant);
1216            return links.Update(updateRequest, handler);
1217        }
1218
1219        public static void EnforceResetValues<TLinkAddress>(this ILinks<TLinkAddress> links,
        ↪  TLinkAddress linkIndex) => links.EnforceResetValues(linkIndex, null);
1220
1221
1222        // TODO: Create a universal version of this method in Platform.Data (with using of for
        ↪  loop)
1223        /// <summary>
1224        /// <para>
1225        /// Enforces the reset values using the specified links.
1226        /// </para>
1227        /// <para></para>
1228        /// </summary>
1229        /// <typeparam name="TLinkAddress">
1230        /// <para>The link.</para>
1231        /// <para></para>
1232        /// </typeparam>
1233        /// <param name="links">
1234        /// <para>The links.</para>
1235        /// <para></para>
1236        /// </param>
1237        /// <param name="linkIndex">
1238        /// <para>The link index.</para>
1239        /// <para></para>
1240        /// </param>
1241        [MethodImpl(MethodImplOptions.AggressiveInlining)]
1242        public static TLinkAddress EnforceResetValues<TLinkAddress>(this ILinks<TLinkAddress>
        ↪  links, TLinkAddress linkIndex, WriteHandler<TLinkAddress>? handler)
1243        {
1244            if (!links.AreValuesReset(linkIndex))
1245            {
1246                return links.ResetValues(linkIndex, handler);
1247            }
1248            return links.Constants.Continue;
1249        }
1250
1251        public static void MergeUsages<TLinkAddress>(this ILinks<TLinkAddress> links,
        ↪  TLinkAddress oldLinkIndex, TLinkAddress newLinkIndex) =>
        ↪  links.MergeUsages(oldLinkIndex, newLinkIndex, null);
1252
1253        /// <summary>
1254        /// Merging two usages graphs, all children of old link moved to be children of new link
        ↪  or deleted.
1255        /// </summary>
1256        [MethodImpl(MethodImplOptions.AggressiveInlining)]
1257        public static TLinkAddress MergeUsages<TLinkAddress>(this ILinks<TLinkAddress> links,
        ↪  TLinkAddress oldLinkIndex, TLinkAddress newLinkIndex, WriteHandler<TLinkAddress>?
        ↪  handler)
1258        {
1259            var equalityComparer = EqualityComparer<TLinkAddress>.Default;
1260            if (equalityComparer.Equals(oldLinkIndex, newLinkIndex))
1261            {
1262                return newLinkIndex;
1263            }
1264            var constants = links.Constants;
1265            var usagesAsSource = links.All(new Link<TLinkAddress>(constants.Any, oldLinkIndex,
            ↪  constants.Any));
1266            WriteHandlerState<TLinkAddress> handlerState = new(constants.Continue,
            ↪  constants.Break, handler);
1267            for (var i = 0; i < usagesAsSource.Count; i++)
1268            {
1269                var usageAsSource = usagesAsSource[i];
1270                if (equalityComparer.Equals(usageAsSource[constants.IndexPart], oldLinkIndex))
1271                {
1272                    continue;
1273                }
1274                var restriction = new
                ↪  LinkAddress<TLinkAddress>(usageAsSource[constants.IndexPart]);
1275                var substitution = new Link<TLinkAddress>(newLinkIndex,
                ↪  usageAsSource[constants.TargetPart]);
1276                handlerState.Apply(links.Update(restriction, substitution,
                ↪  handlerState.Handler));
1277            }
```

```
1278            var usagesAsTarget = links.All(new Link<TLinkAddress>(constants.Any, constants.Any,
                ↪ oldLinkIndex));
1279            for (var i = 0; i < usagesAsTarget.Count; i++)
1280            {
1281                var usageAsTarget = usagesAsTarget[i];
1282                if (equalityComparer.Equals(usageAsTarget[constants.IndexPart], oldLinkIndex))
1283                {
1284                    continue;
1285                }
1286                var restriction = links.GetLink(usageAsTarget[constants.IndexPart]);
1287                var substitution = new Link<TLinkAddress>(usageAsTarget[constants.TargetPart],
                    ↪ newLinkIndex);
1288                handlerState.Apply(links.Update(restriction, substitution,
                    ↪ handlerState.Handler));
1289            }
1290            return handlerState.Result;
1291        }
1292
1293        public static TLinkAddress MergeAndDelete<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪ TLinkAddress oldLinkIndex, TLinkAddress newLinkIndex)
1294        {
1295            var equalityComparer = EqualityComparer<TLinkAddress>.Default;
1296            if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
1297            {
1298                links.MergeUsages(oldLinkIndex, newLinkIndex);
1299                links.Delete(oldLinkIndex);
1300            }
1301            return newLinkIndex;
1302        }
1303
1304        /// <summary>
1305        /// Replace one link with another (replaced link is deleted, children are updated or
            ↪ deleted).
1306        /// </summary>
1307        [MethodImpl(MethodImplOptions.AggressiveInlining)]
1308        public static TLinkAddress MergeAndDelete<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪ TLinkAddress oldLinkIndex, TLinkAddress newLinkIndex, WriteHandler<TLinkAddress>?
            ↪ handler)
1309        {
1310            var equalityComparer = EqualityComparer<TLinkAddress>.Default;
1311            var constants = links.Constants;
1312            WriteHandlerState<TLinkAddress> handlerState = new(constants.Continue,
                ↪ constants.Break, handler);
1313            if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
1314            {
1315                handlerState.Apply(links.MergeUsages(oldLinkIndex, newLinkIndex,
                    ↪ handlerState.Handler));
1316                handlerState.Apply(links.Delete(oldLinkIndex, handlerState.Handler));
1317            }
1318            return handlerState.Result;
1319        }
1320
1321        /// <summary>
1322        /// <para>
1323        /// Decorates the with automatic uniqueness and usages resolution using the specified
            ↪ links.
1324        /// </para>
1325        /// <para></para>
1326        /// </summary>
1327        /// <typeparam name="TLinkAddress">
1328        /// <para>The link.</para>
1329        /// <para></para>
1330        /// </typeparam>
1331        /// <param name="links">
1332        /// <para>The links.</para>
1333        /// <para></para>
1334        /// </param>
1335        /// <returns>
1336        /// <para>The links.</para>
1337        /// <para></para>
1338        /// </returns>
1339        [MethodImpl(MethodImplOptions.AggressiveInlining)]
1340        public static ILinks<TLinkAddress>
            ↪ DecorateWithAutomaticUniquenessAndUsagesResolution<TLinkAddress>(this
            ↪ ILinks<TLinkAddress> links)
1341        {
1342            links = new LinksCascadeUsagesResolver<TLinkAddress>(links);
1343            links = new NonNullContentsLinkDeletionResolver<TLinkAddress>(links);
```

```
            links = new LinksCascadeUniquenessAndUsagesResolver<TLinkAddress>(links);
            return links;
        }

        /// <summary>
        /// <para>
        /// Formats the links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TLinkAddress">
        /// <para>The link.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="links">
        /// <para>The links.</para>
        /// <para></para>
        /// </param>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string Format<TLinkAddress>(this ILinks<TLinkAddress> links,
        ↪  IList<TLinkAddress>? link)
        {
            var constants = links.Constants;
            return $"({link[constants.IndexPart]}: {link[constants.SourcePart]}
            ↪  {link[constants.TargetPart]})";
        }

        /// <summary>
        /// <para>
        /// Formats the links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TLinkAddress">
        /// <para>The link.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="links">
        /// <para>The links.</para>
        /// <para></para>
        /// </param>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string Format<TLinkAddress>(this ILinks<TLinkAddress> links, TLinkAddress
        ↪  link) => links.Format(links.GetLink(link));
    }
}
```

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

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

namespace Platform.Data.Doublets
{
    /// <summary>
    /// <para>
    /// Defines the synchronized links.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ISynchronizedLinks{TLinkAddress, ILinks{TLinkAddress},
    ↪  LinksConstants{TLinkAddress}}"/>
    /// <seealso cref="ILinks{TLinkAddress}"/>
```

```
    public interface ISynchronizedLinks<TLinkAddress> : ISynchronizedLinks<TLinkAddress,
    ↪  ILinks<TLinkAddress>, LinksConstants<TLinkAddress>>, ILinks<TLinkAddress>
    {
    }
}
```

1.25 ./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<TLinkAddress> : IEquatable<Link<TLinkAddress>>,
    ↪  IReadOnlyList<TLinkAddress>, IList<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// The link.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly Link<TLinkAddress> Null = new Link<TLinkAddress>();
        private static readonly LinksConstants<TLinkAddress> _constants =
        ↪  Default<LinksConstants<TLinkAddress>>.Instance;
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;
        private const int Length = 3;

        /// <summary>
        /// <para>
        /// The index.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly TLinkAddress Index;
        /// <summary>
        /// <para>
        /// The source.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly TLinkAddress Source;
        /// <summary>
        /// <para>
        /// The target.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly TLinkAddress Target;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Link"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="values">
        /// <para>A values.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Link(params TLinkAddress[] values) => SetValues(values, out Index, out Source,
        ↪  out Target);

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Link"/> instance.
        /// </para>
```

```
69          /// <para></para>
70          /// </summary>
71          /// <param name="values">
72          /// <para>A values.</para>
73          /// <para></para>
74          /// </param>
75          [MethodImpl(MethodImplOptions.AggressiveInlining)]
76          public Link(IList<TLinkAddress>? values) => SetValues(values, out Index, out Source, out
            ↪  Target);
77
78          /// <summary>
79          /// <para>
80          /// Initializes a new <see cref="Link"/> instance.
81          /// </para>
82          /// <para></para>
83          /// </summary>
84          /// <param name="other">
85          /// <para>A other.</para>
86          /// <para></para>
87          /// </param>
88          /// <exception cref="NotSupportedException">
89          /// <para></para>
90          /// <para></para>
91          /// </exception>
92          [MethodImpl(MethodImplOptions.AggressiveInlining)]
93          public Link(object other)
94          {
95              if (other is Link<TLinkAddress> otherLink)
96              {
97                  SetValues(ref otherLink, out Index, out Source, out Target);
98              }
99              else if(other is IList<TLinkAddress> otherList)
100             {
101                 SetValues(otherList, out Index, out Source, out Target);
102             }
103             else
104             {
105                 throw new NotSupportedException();
106             }
107         }
108
109         /// <summary>
110         /// <para>
111         /// Initializes a new <see cref="Link"/> instance.
112         /// </para>
113         /// <para></para>
114         /// </summary>
115         /// <param name="other">
116         /// <para>A other.</para>
117         /// <para></para>
118         /// </param>
119         [MethodImpl(MethodImplOptions.AggressiveInlining)]
120         public Link(ref Link<TLinkAddress> other) => SetValues(ref other, out Index, out Source,
            ↪  out Target);
121
122         /// <summary>
123         /// <para>
124         /// Initializes a new <see cref="Link"/> instance.
125         /// </para>
126         /// <para></para>
127         /// </summary>
128         /// <param name="index">
129         /// <para>A index.</para>
130         /// <para></para>
131         /// </param>
132         /// <param name="source">
133         /// <para>A source.</para>
134         /// <para></para>
135         /// </param>
136         /// <param name="target">
137         /// <para>A target.</para>
138         /// <para></para>
139         /// </param>
140         [MethodImpl(MethodImplOptions.AggressiveInlining)]
141         public Link(TLinkAddress index, TLinkAddress source, TLinkAddress target)
142         {
143             Index = index;
144             Source = source;
```

```
145             Target = target;
146         }
147         [MethodImpl(MethodImplOptions.AggressiveInlining)]
148         private static void SetValues(ref Link<TLinkAddress> other, out TLinkAddress index, out
            ↪ TLinkAddress source, out TLinkAddress target)
149         {
150             index = other.Index;
151             source = other.Source;
152             target = other.Target;
153         }
154         [MethodImpl(MethodImplOptions.AggressiveInlining)]
155         private static void SetValues(IList<TLinkAddress>? values, out TLinkAddress index, out
            ↪ TLinkAddress source, out TLinkAddress target)
156         {
157             if (values == null)
158             {
159                 index = default;
160                 source = default;
161                 target = default;
162                 return;
163             }
164             switch (values.Count)
165             {
166                 case 3:
167                     index = values[0];
168                     source = values[1];
169                     target = values[2];
170                     break;
171                 case 2:
172                     index = values[0];
173                     source = values[1];
174                     target = default;
175                     break;
176                 case 1:
177                     index = values[0];
178                     source = default;
179                     target = default;
180                     break;
181                 default:
182                     index = default;
183                     source = default;
184                     target = default;
185                     break;
186             }
187         }
188 
189         /// <summary>
190         /// <para>
191         /// Gets the hash code.
192         /// </para>
193         /// <para></para>
194         /// </summary>
195         /// <returns>
196         /// <para>The int</para>
197         /// <para></para>
198         /// </returns>
199         [MethodImpl(MethodImplOptions.AggressiveInlining)]
200         public override int GetHashCode() => (Index, Source, Target).GetHashCode();
201 
202         /// <summary>
203         /// <para>
204         /// Determines whether this instance is null.
205         /// </para>
206         /// <para></para>
207         /// </summary>
208         /// <returns>
209         /// <para>The bool</para>
210         /// <para></para>
211         /// </returns>
212         [MethodImpl(MethodImplOptions.AggressiveInlining)]
213         public bool IsNull() => _equalityComparer.Equals(Index, _constants.Null)
214                                 && _equalityComparer.Equals(Source, _constants.Null)
215                                 && _equalityComparer.Equals(Target, _constants.Null);
216 
217         /// <summary>
218         /// <para>
219         /// Determines whether this instance equals.
220         /// </para>
221         /// <para></para>
```

```
222         /// </summary>
223         /// <param name="other">
224         /// <para>The other.</para>
225         /// <para></para>
226         /// </param>
227         /// <returns>
228         /// <para>The bool</para>
229         /// <para></para>
230         /// </returns>
231         [MethodImpl(MethodImplOptions.AggressiveInlining)]
232         public override bool Equals(object other) => other is Link<TLinkAddress> &&
            ↪ Equals((Link<TLinkAddress>)other);
233
234         /// <summary>
235         /// <para>
236         /// Determines whether this instance equals.
237         /// </para>
238         /// <para></para>
239         /// </summary>
240         /// <param name="other">
241         /// <para>The other.</para>
242         /// <para></para>
243         /// </param>
244         /// <returns>
245         /// <para>The bool</para>
246         /// <para></para>
247         /// </returns>
248         [MethodImpl(MethodImplOptions.AggressiveInlining)]
249         public bool Equals(Link<TLinkAddress> other) => _equalityComparer.Equals(Index,
            ↪ other.Index)
250                                                 && _equalityComparer.Equals(Source, other.Source)
251                                                 && _equalityComparer.Equals(Target, other.Target);
252
253         /// <summary>
254         /// <para>
255         /// Returns the string using the specified index.
256         /// </para>
257         /// <para></para>
258         /// </summary>
259         /// <param name="index">
260         /// <para>The index.</para>
261         /// <para></para>
262         /// </param>
263         /// <param name="source">
264         /// <para>The source.</para>
265         /// <para></para>
266         /// </param>
267         /// <param name="target">
268         /// <para>The target.</para>
269         /// <para></para>
270         /// </param>
271         /// <returns>
272         /// <para>The string</para>
273         /// <para></para>
274         /// </returns>
275         [MethodImpl(MethodImplOptions.AggressiveInlining)]
276         public static string ToString(TLinkAddress index, TLinkAddress source, TLinkAddress
            ↪ target) => $"({index}: {source}->{target})";
277
278         /// <summary>
279         /// <para>
280         /// Returns the string using the specified source.
281         /// </para>
282         /// <para></para>
283         /// </summary>
284         /// <param name="source">
285         /// <para>The source.</para>
286         /// <para></para>
287         /// </param>
288         /// <param name="target">
289         /// <para>The target.</para>
290         /// <para></para>
291         /// </param>
292         /// <returns>
293         /// <para>The string</para>
294         /// <para></para>
295         /// </returns>
```

```
296             [MethodImpl(MethodImplOptions.AggressiveInlining)]
297             public static string ToString(TLinkAddress source, TLinkAddress target) =>
                ↪  $"({source}->{target})";
298
299             [MethodImpl(MethodImplOptions.AggressiveInlining)]
300             public static implicit operator TLinkAddress[](Link<TLinkAddress> link) =>
                ↪  link.ToArray();
301
302             [MethodImpl(MethodImplOptions.AggressiveInlining)]
303             public static implicit operator Link<TLinkAddress>(TLinkAddress[] linkArray) => new
                ↪  Link<TLinkAddress>(linkArray);
304
305             /// <summary>
306             /// <para>
307             /// Returns the string.
308             /// </para>
309             /// <para></para>
310             /// </summary>
311             /// <returns>
312             /// <para>The string</para>
313             /// <para></para>
314             /// </returns>
315             [MethodImpl(MethodImplOptions.AggressiveInlining)]
316             public override string ToString() => _equalityComparer.Equals(Index, _constants.Null) ?
                ↪  ToString(Source, Target) : ToString(Index, Source, Target);
317
318             #region IList
319
320             /// <summary>
321             /// <para>
322             /// Gets the count value.
323             /// </para>
324             /// <para></para>
325             /// </summary>
326             public int Count
327             {
328                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
329                 get => Length;
330             }
331
332             /// <summary>
333             /// <para>
334             /// Gets the is read only value.
335             /// </para>
336             /// <para></para>
337             /// </summary>
338             public bool IsReadOnly
339             {
340                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
341                 get => true;
342             }
343
344             /// <summary>
345             /// <para>
346             /// The not supported exception.
347             /// </para>
348             /// <para></para>
349             /// </summary>
350             public TLinkAddress this[int index]
351             {
352                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
353                 get
354                 {
355                     Ensure.OnDebug.ArgumentInRange(index, new Range<int>(0, Length - 1),
                        ↪  nameof(index));
356                     if (index == _constants.IndexPart)
357                     {
358                         return Index;
359                     }
360                     if (index == _constants.SourcePart)
361                     {
362                         return Source;
363                     }
364                     if (index == _constants.TargetPart)
365                     {
366                         return Target;
367                     }
368                     throw new NotSupportedException(); // Impossible path due to
                        ↪  Ensure.ArgumentInRange
```

```
            }
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set => throw new NotSupportedException();
        }

        /// <summary>
        /// <para>
        /// Gets the enumerator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The enumerator</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

        /// <summary>
        /// <para>
        /// Gets the enumerator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>An enumerator of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IEnumerator<TLinkAddress> GetEnumerator()
        {
            yield return Index;
            yield return Source;
            yield return Target;
        }

        /// <summary>
        /// <para>
        /// Adds the item.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="item">
        /// <para>The item.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Add(TLinkAddress item) => throw new NotSupportedException();

        /// <summary>
        /// <para>
        /// Clears this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Clear() => throw new NotSupportedException();

        /// <summary>
        /// <para>
        /// Determines whether this instance contains.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="item">
        /// <para>The item.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Contains(TLinkAddress item) => IndexOf(item) >= 0;

        /// <summary>
        /// <para>
        /// Copies the to using the specified array.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="array">
        /// <para>The array.</para>
        /// <para></para>
        /// </param>
        /// <param name="arrayIndex">
        /// <para>The array index.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="InvalidOperationException">
        /// <para></para>
        /// <para></para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void CopyTo(TLinkAddress[] 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;
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance remove.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="item">
        /// <para>The item.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Remove(TLinkAddress item) => Throw.A.NotSupportedExceptionAndReturn<bool>();

        /// <summary>
        /// <para>
        /// Indexes the of using the specified item.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="item">
        /// <para>The item.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The int</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public int IndexOf(TLinkAddress 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;
        }
```

```
524
525         /// <summary>
526         /// <para>
527         /// Inserts the index.
528         /// </para>
529         /// <para></para>
530         /// </summary>
531         /// <param name="index">
532         /// <para>The index.</para>
533         /// <para></para>
534         /// </param>
535         /// <param name="item">
536         /// <para>The item.</para>
537         /// <para></para>
538         /// </param>
539         [MethodImpl(MethodImplOptions.AggressiveInlining)]
540         public void Insert(int index, TLinkAddress item) => throw new NotSupportedException();
541
542         /// <summary>
543         /// <para>
544         /// Removes the at using the specified index.
545         /// </para>
546         /// <para></para>
547         /// </summary>
548         /// <param name="index">
549         /// <para>The index.</para>
550         /// <para></para>
551         /// </param>
552         [MethodImpl(MethodImplOptions.AggressiveInlining)]
553         public void RemoveAt(int index) => throw new NotSupportedException();
554
555         [MethodImpl(MethodImplOptions.AggressiveInlining)]
556         public static bool operator ==(Link<TLinkAddress> left, Link<TLinkAddress> right) =>
            ↪  left.Equals(right);
557
558         [MethodImpl(MethodImplOptions.AggressiveInlining)]
559         public static bool operator !=(Link<TLinkAddress> left, Link<TLinkAddress> right) =>
            ↪  !(left == right);
560
561         #endregion
562     }
563 }
```

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

```
1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets
6  {
7      /// <summary>
8      /// <para>
9      /// Represents the link extensions.
10     /// </para>
11     /// <para></para>
12     /// </summary>
13     public static class LinkExtensions
14     {
15         /// <summary>
16         /// <para>
17         /// Determines whether is full point.
18         /// </para>
19         /// <para></para>
20         /// </summary>
21         /// <typeparam name="TLinkAddress">
22         /// <para>The link.</para>
23         /// <para></para>
24         /// </typeparam>
25         /// <param name="link">
26         /// <para>The link.</para>
27         /// <para></para>
28         /// </param>
29         /// <returns>
30         /// <para>The bool</para>
31         /// <para></para>
32         /// </returns>
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public static bool IsFullPoint<TLinkAddress>(this Link<TLinkAddress> link) =>
            ↪  Point<TLinkAddress>.IsFullPoint(link);
```

```

        /// <summary>
        /// <para>
        /// Determines whether is partial point.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TLinkAddress">
        /// <para>The link.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsPartialPoint<TLinkAddress>(this Link<TLinkAddress> link) =>
        ↪  Point<TLinkAddress>.IsPartialPoint(link);
    }
}
```

1.27 ./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
{
    /// <summary>
    /// <para>
    /// Represents the links operator base.
    /// </para>
    /// <para></para>
    /// </summary>
    public abstract class LinksOperatorBase<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// The links.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly ILinks<TLinkAddress> _links;

        /// <summary>
        /// <para>
        /// Gets the links value.
        /// </para>
        /// <para></para>
        /// </summary>
        public ILinks<TLinkAddress> Links
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _links;
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksOperatorBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected LinksOperatorBase(ILinks<TLinkAddress> links) => _links = links;
    }
}
```

1.28 ./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
{
    /// <summary>
    /// <para>
    /// Defines the links list methods.
    /// </para>
    /// <para></para>
    /// </summary>
    public interface ILinksListMethods<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Detaches the free link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="freeLink">
        /// <para>The free link.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        void Detach(TLinkAddress freeLink);

        /// <summary>
        /// <para>
        /// Attaches the as first using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        void AttachAsFirst(TLinkAddress link);
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Memory
{
    /// <summary>
    /// <para>
    /// Defines the links tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    public interface ILinksTreeMethods<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Counts the usages using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        TLinkAddress CountUsages(TLinkAddress root);

        /// <summary>
        /// <para>
        /// Searches the source.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        TLinkAddress Search(TLinkAddress source, TLinkAddress target);

        /// <summary>
        /// <para>
        /// Eaches the usage using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="handler">
        /// <para>The handler.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        TLinkAddress EachUsage(TLinkAddress root, ReadHandler<TLinkAddress>? handler);

        /// <summary>
        /// <para>
        /// Detaches the root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="linkIndex">
        /// <para>The link index.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        void Detach(ref TLinkAddress root, TLinkAddress linkIndex);

        /// <summary>
        /// <para>
        /// Attaches the root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="linkIndex">
        /// <para>The link index.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        void Attach(ref TLinkAddress root, TLinkAddress linkIndex);
    }
}
```

1.30 ./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
```

```
{
    /// <summary>
    /// <para>
    /// The index tree type enum.
    /// </para>
    /// <para></para>
    /// </summary>
    public enum IndexTreeType
    {
        /// <summary>
        /// <para>
        /// The default index tree type.
        /// </para>
        /// <para></para>
        /// </summary>
        Default = 0,
        /// <summary>
        /// <para>
        /// The size balanced tree index tree type.
        /// </para>
        /// <para></para>
        /// </summary>
        SizeBalancedTree = 1,
        /// <summary>
        /// <para>
        /// The recursionless size balanced tree index tree type.
        /// </para>
        /// <para></para>
        /// </summary>
        RecursionlessSizeBalancedTree = 2,
        /// <summary>
        /// <para>
        /// The sized and threaded avl balanced tree index tree type.
        /// </para>
        /// <para></para>
        /// </summary>
        SizedAndThreadedAVLBalancedTree = 3
    }
}
```

1.31 ./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
{
    /// <summary>
    /// <para>
    /// The links header.
    /// </para>
    /// <para></para>
    /// </summary>
    public struct LinksHeader<TLinkAddress> : IEquatable<LinksHeader<TLinkAddress>>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;

        /// <summary>
        /// <para>
        /// The size.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly long SizeInBytes = Structure<LinksHeader<TLinkAddress>>.Size;

        /// <summary>
        /// <para>
        /// The allocated links.
        /// </para>
        /// <para></para>
        /// </summary>
        public TLinkAddress AllocatedLinks;
        /// <summary>
        /// <para>
        /// The reserved links.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        public TLinkAddress ReservedLinks;
        /// <summary>
        /// <para>
        /// The free links.
        /// </para>
        /// <para></para>
        /// </summary>
        public TLinkAddress FreeLinks;
        /// <summary>
        /// <para>
        /// The first free link.
        /// </para>
        /// <para></para>
        /// </summary>
        public TLinkAddress FirstFreeLink;
        /// <summary>
        /// <para>
        /// The root as source.
        /// </para>
        /// <para></para>
        /// </summary>
        public TLinkAddress RootAsSource;
        /// <summary>
        /// <para>
        /// The root as target.
        /// </para>
        /// <para></para>
        /// </summary>
        public TLinkAddress RootAsTarget;
        /// <summary>
        /// <para>
        /// The last free link.
        /// </para>
        /// <para></para>
        /// </summary>
        public TLinkAddress LastFreeLink;
        /// <summary>
        /// <para>
        /// The reserved.
        /// </para>
        /// <para></para>
        /// </summary>
        public TLinkAddress Reserved8;

        /// <summary>
        /// <para>
        /// Determines whether this instance equals.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="obj">
        /// <para>The obj.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Equals(object obj) => obj is LinksHeader<TLinkAddress> linksHeader
        ↪  ? Equals(linksHeader) : false;

        /// <summary>
        /// <para>
        /// Determines whether this instance equals.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="other">
        /// <para>The other.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
```

```
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Equals(LinksHeader<TLinkAddress> 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);

        /// <summary>
        /// <para>
        /// Gets the hash code.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The int</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override int GetHashCode() => (AllocatedLinks, ReservedLinks, FreeLinks,
        ↪  FirstFreeLink, RootAsSource, RootAsTarget, LastFreeLink, Reserved8).GetHashCode();

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

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

1.32 ./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 Platform.Delegates;
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
{
    /// <summary>
    /// <para>
    /// Represents the external links recursionless size balanced tree methods base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="RecursionlessSizeBalancedTreeMethods{TLinkAddress}"/>
    /// <seealso cref="ILinksTreeMethods{TLinkAddress}"/>
    public unsafe abstract class
    ↪  ExternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLinkAddress> :
    ↪  RecursionlessSizeBalancedTreeMethods<TLinkAddress>, ILinksTreeMethods<TLinkAddress>
    {
        private static readonly UncheckedConverter<TLinkAddress, long> _addressToInt64Converter
        ↪  = UncheckedConverter<TLinkAddress, long>.Default;

        /// <summary>
        /// <para>
        /// The break.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly TLinkAddress Break;
        /// <summary>
        /// <para>
        /// The continue.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly TLinkAddress Continue;
```

```
        /// <summary>
        /// <para>
        /// The links data parts.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly byte* LinksDataParts;
        /// <summary>
        /// <para>
        /// The links index parts.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly byte* LinksIndexParts;
        /// <summary>
        /// <para>
        /// The header.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly byte* Header;

        /// <summary>
        /// <para>
        /// Initializes a new <see
        ↪  cref="ExternalLinksRecursionlessSizeBalancedTreeMethodsBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksDataParts">
        /// <para>A links data parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksIndexParts">
        /// <para>A links index parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected
        ↪  ExternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLinkAddress>
        ↪  constants, byte* linksDataParts, byte* linksIndexParts, byte* header)
        {
            LinksDataParts = linksDataParts;
            LinksIndexParts = linksIndexParts;
            Header = header;
            Break = constants.Break;
            Continue = constants.Continue;
        }

        /// <summary>
        /// <para>
        /// Gets the tree root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TLinkAddress GetTreeRoot();

        /// <summary>
        /// <para>
        /// Gets the base part value using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
```

```
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TLinkAddress GetBasePartValue(TLinkAddress link);

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the right of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <param name="rootSource">
        /// <para>The root source.</para>
        /// <para></para>
        /// </param>
        /// <param name="rootTarget">
        /// <para>The root target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract bool FirstIsToTheRightOfSecond(TLinkAddress source, TLinkAddress
        ↪  target, TLinkAddress rootSource, TLinkAddress rootTarget);

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <param name="rootSource">
        /// <para>The root source.</para>
        /// <para></para>
        /// </param>
        /// <param name="rootTarget">
        /// <para>The root target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract bool FirstIsToTheLeftOfSecond(TLinkAddress source, TLinkAddress
        ↪  target, TLinkAddress rootSource, TLinkAddress rootTarget);

        /// <summary>
        /// <para>
        /// Gets the header reference.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A ref links header of t link</para>
        /// <para></para>
```

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

        /// <summary>
        /// <para>
        /// Gets the link data part reference using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link data part of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref RawLinkDataPart<TLinkAddress>
        ↪  GetLinkDataPartReference(TLinkAddress link) => ref
        ↪  AsRef<RawLinkDataPart<TLinkAddress>>(LinksDataParts +
        ↪  (RawLinkDataPart<TLinkAddress>.SizeInBytes *
        ↪  _addressToInt64Converter.Convert(link)));

        /// <summary>
        /// <para>
        /// Gets the link index part reference using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link index part of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref RawLinkIndexPart<TLinkAddress>
        ↪  GetLinkIndexPartReference(TLinkAddress link) => ref
        ↪  AsRef<RawLinkIndexPart<TLinkAddress>>(LinksIndexParts +
        ↪  (RawLinkIndexPart<TLinkAddress>.SizeInBytes *
        ↪  _addressToInt64Converter.Convert(link)));

        /// <summary>
        /// <para>
        /// Gets the link values using the specified link index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="linkIndex">
        /// <para>The link index.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A list of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual IList<TLinkAddress>? GetLinkValues(TLinkAddress linkIndex)
        {
            ref var link = ref GetLinkDataPartReference(linkIndex);
            return new Link<TLinkAddress>(linkIndex, link.Source, link.Target);
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
```

```
260         /// <param name="second">
261         /// <para>The second.</para>
262         /// <para></para>
263         /// </param>
264         /// <returns>
265         /// <para>The bool</para>
266         /// <para></para>
267         /// </returns>
268         [MethodImpl(MethodImplOptions.AggressiveInlining)]
269         protected override bool FirstIsToTheLeftOfSecond(TLinkAddress first, TLinkAddress second)
270         {
271             ref var firstLink = ref GetLinkDataPartReference(first);
272             ref var secondLink = ref GetLinkDataPartReference(second);
273             return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
274         }
275 
276         /// <summary>
277         /// <para>
278         /// Determines whether this instance first is to the right of second.
279         /// </para>
280         /// <para></para>
281         /// </summary>
282         /// <param name="first">
283         /// <para>The first.</para>
284         /// <para></para>
285         /// </param>
286         /// <param name="second">
287         /// <para>The second.</para>
288         /// <para></para>
289         /// </param>
290         /// <returns>
291         /// <para>The bool</para>
292         /// <para></para>
293         /// </returns>
294         [MethodImpl(MethodImplOptions.AggressiveInlining)]
295         protected override bool FirstIsToTheRightOfSecond(TLinkAddress first, TLinkAddress
                ↪  second)
296         {
297             ref var firstLink = ref GetLinkDataPartReference(first);
298             ref var secondLink = ref GetLinkDataPartReference(second);
299             return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
300         }
301 
302         /// <summary>
303         /// <para>
304         /// The zero.
305         /// </para>
306         /// <para></para>
307         /// </summary>
308         public TLinkAddress this[TLinkAddress index]
309         {
310             [MethodImpl(MethodImplOptions.AggressiveInlining)]
311             get
312             {
313                 var root = GetTreeRoot();
314                 if (GreaterOrEqualThan(index, GetSize(root)))
315                 {
316                     return Zero;
317                 }
318                 while (!EqualToZero(root))
319                 {
320                     var left = GetLeftOrDefault(root);
321                     var leftSize = GetSizeOrZero(left);
322                     if (LessThan(index, leftSize))
323                     {
324                         root = left;
325                         continue;
326                     }
327                     if (AreEqual(index, leftSize))
328                     {
329                         return root;
330                     }
331                     root = GetRightOrDefault(root);
332                     index = Subtract(index, Increment(leftSize));
333                 }
```

```
                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 TLinkAddress Search(TLinkAddress source, TLinkAddress 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
        /// <summary>
        /// <para>
        /// Counts the usages using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress CountUsages(TLinkAddress 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);
        }

        /// <summary>
        /// <para>
        /// Eaches the usage using the specified base.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="@base">
        /// <para>The base.</para>
        /// <para></para>
        /// </param>
        /// <param name="handler">
        /// <para>The handler.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress EachUsage(TLinkAddress @base, ReadHandler<TLinkAddress>? 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 TLinkAddress EachUsageCore(TLinkAddress @base, TLinkAddress link,
        ↪  ReadHandler<TLinkAddress>? 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;
```

```
        }

        /// <summary>
        /// <para>
        /// Prints the node value using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="sb">
        /// <para>The sb.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void PrintNodeValue(TLinkAddress node, StringBuilder sb)
        {
            ref var link = ref GetLinkDataPartReference(node);
            sb.Append(' ');
            sb.Append(link.Source);
            sb.Append('-');
            sb.Append('>');
            sb.Append(link.Target);
        }
    }
}
```

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

```
using System;
using System.Text;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections.Methods.Trees;
using Platform.Converters;
using Platform.Delegates;
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
{
    /// <summary>
    /// <para>
    /// Represents the external links size balanced tree methods base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="SizeBalancedTreeMethods{TLinkAddress}"/>
    /// <seealso cref="ILinksTreeMethods{TLinkAddress}"/>
    public unsafe abstract class ExternalLinksSizeBalancedTreeMethodsBase<TLinkAddress> :
    ↪  SizeBalancedTreeMethods<TLinkAddress>, ILinksTreeMethods<TLinkAddress>
    {
        private static readonly UncheckedConverter<TLinkAddress, long> _addressToInt64Converter
        ↪  = UncheckedConverter<TLinkAddress, long>.Default;

        /// <summary>
        /// <para>
        /// The break.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly TLinkAddress Break;
        /// <summary>
        /// <para>
        /// The continue.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly TLinkAddress Continue;
        /// <summary>
        /// <para>
        /// The links data parts.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly byte* LinksDataParts;
        /// <summary>
```

```
        /// <para>
        /// The links index parts.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly byte* LinksIndexParts;
        /// <summary>
        /// <para>
        /// The header.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly byte* Header;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="ExternalLinksSizeBalancedTreeMethodsBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksDataParts">
        /// <para>A links data parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksIndexParts">
        /// <para>A links index parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected ExternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLinkAddress>
        ↪  constants, byte* linksDataParts, byte* linksIndexParts, byte* header)
        {
            LinksDataParts = linksDataParts;
            LinksIndexParts = linksIndexParts;
            Header = header;
            Break = constants.Break;
            Continue = constants.Continue;
        }

        /// <summary>
        /// <para>
        /// Gets the tree root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TLinkAddress GetTreeRoot();

        /// <summary>
        /// <para>
        /// Gets the base part value using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TLinkAddress GetBasePartValue(TLinkAddress link);

        /// <summary>
```

```
125         /// <para>
126         /// Determines whether this instance first is to the right of second.
127         /// </para>
128         /// <para></para>
129         /// </summary>
130         /// <param name="source">
131         /// <para>The source.</para>
132         /// <para></para>
133         /// </param>
134         /// <param name="target">
135         /// <para>The target.</para>
136         /// <para></para>
137         /// </param>
138         /// <param name="rootSource">
139         /// <para>The root source.</para>
140         /// <para></para>
141         /// </param>
142         /// <param name="rootTarget">
143         /// <para>The root target.</para>
144         /// <para></para>
145         /// </param>
146         /// <returns>
147         /// <para>The bool</para>
148         /// <para></para>
149         /// </returns>
150         [MethodImpl(MethodImplOptions.AggressiveInlining)]
151         protected abstract bool FirstIsToTheRightOfSecond(TLinkAddress source, TLinkAddress
          ↪  target, TLinkAddress rootSource, TLinkAddress rootTarget);
152
153         /// <summary>
154         /// <para>
155         /// Determines whether this instance first is to the left of second.
156         /// </para>
157         /// <para></para>
158         /// </summary>
159         /// <param name="source">
160         /// <para>The source.</para>
161         /// <para></para>
162         /// </param>
163         /// <param name="target">
164         /// <para>The target.</para>
165         /// <para></para>
166         /// </param>
167         /// <param name="rootSource">
168         /// <para>The root source.</para>
169         /// <para></para>
170         /// </param>
171         /// <param name="rootTarget">
172         /// <para>The root target.</para>
173         /// <para></para>
174         /// </param>
175         /// <returns>
176         /// <para>The bool</para>
177         /// <para></para>
178         /// </returns>
179         [MethodImpl(MethodImplOptions.AggressiveInlining)]
180         protected abstract bool FirstIsToTheLeftOfSecond(TLinkAddress source, TLinkAddress
          ↪  target, TLinkAddress rootSource, TLinkAddress rootTarget);
181
182         /// <summary>
183         /// <para>
184         /// Gets the header reference.
185         /// </para>
186         /// <para></para>
187         /// </summary>
188         /// <returns>
189         /// <para>A ref links header of t link</para>
190         /// <para></para>
191         /// </returns>
192         [MethodImpl(MethodImplOptions.AggressiveInlining)]
193         protected virtual ref LinksHeader<TLinkAddress> GetHeaderReference() => ref
          ↪  AsRef<LinksHeader<TLinkAddress>>(Header);
194
195         /// <summary>
196         /// <para>
197         /// Gets the link data part reference using the specified link.
198         /// </para>
199         /// <para></para>
```

```
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link data part of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref RawLinkDataPart<TLinkAddress>
        ↪  GetLinkDataPartReference(TLinkAddress link) => ref
        ↪  AsRef<RawLinkDataPart<TLinkAddress>>(LinksDataParts +
        ↪  (RawLinkDataPart<TLinkAddress>.SizeInBytes *
        ↪  _addressToInt64Converter.Convert(link)));

        /// <summary>
        /// <para>
        /// Gets the link index part reference using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link index part of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref RawLinkIndexPart<TLinkAddress>
        ↪  GetLinkIndexPartReference(TLinkAddress link) => ref
        ↪  AsRef<RawLinkIndexPart<TLinkAddress>>(LinksIndexParts +
        ↪  (RawLinkIndexPart<TLinkAddress>.SizeInBytes *
        ↪  _addressToInt64Converter.Convert(link)));

        /// <summary>
        /// <para>
        /// Gets the link values using the specified link index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="linkIndex">
        /// <para>The link index.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A list of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual IList<TLinkAddress>? GetLinkValues(TLinkAddress linkIndex)
        {
            ref var link = ref GetLinkDataPartReference(linkIndex);
            return new Link<TLinkAddress>(linkIndex, link.Source, link.Target);
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLinkAddress first, TLinkAddress second)
```

```
270         {
271             ref var firstLink = ref GetLinkDataPartReference(first);
272             ref var secondLink = ref GetLinkDataPartReference(second);
273             return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
274         }
275 
276         /// <summary>
277         /// <para>
278         /// Determines whether this instance first is to the right of second.
279         /// </para>
280         /// <para></para>
281         /// </summary>
282         /// <param name="first">
283         /// <para>The first.</para>
284         /// <para></para>
285         /// </param>
286         /// <param name="second">
287         /// <para>The second.</para>
288         /// <para></para>
289         /// </param>
290         /// <returns>
291         /// <para>The bool</para>
292         /// <para></para>
293         /// </returns>
294         [MethodImpl(MethodImplOptions.AggressiveInlining)]
295         protected override bool FirstIsToTheRightOfSecond(TLinkAddress first, TLinkAddress
                ↪  second)
296         {
297             ref var firstLink = ref GetLinkDataPartReference(first);
298             ref var secondLink = ref GetLinkDataPartReference(second);
299             return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
300         }
301 
302         /// <summary>
303         /// <para>
304         /// The zero.
305         /// </para>
306         /// <para></para>
307         /// </summary>
308         public TLinkAddress this[TLinkAddress index]
309         {
310             [MethodImpl(MethodImplOptions.AggressiveInlining)]
311             get
312             {
313                 var root = GetTreeRoot();
314                 if (GreaterOrEqualThan(index, GetSize(root)))
315                 {
316                     return Zero;
317                 }
318                 while (!EqualToZero(root))
319                 {
320                     var left = GetLeftOrDefault(root);
321                     var leftSize = GetSizeOrZero(left);
322                     if (LessThan(index, leftSize))
323                     {
324                         root = left;
325                         continue;
326                     }
327                     if (AreEqual(index, leftSize))
328                     {
329                         return root;
330                     }
331                     root = GetRightOrDefault(root);
332                     index = Subtract(index, Increment(leftSize));
333                 }
334                 return Zero; // TODO: Impossible situation exception (only if tree structure
                    ↪  broken)
335             }
336         }
337 
338         /// <summary>
339         /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
                ↪  (концом).
340         /// </summary>
341         /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
342         /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
```

```
343             /// <returns>Индекс искомой связи.</returns>
344             [MethodImpl(MethodImplOptions.AggressiveInlining)]
345             public TLinkAddress Search(TLinkAddress source, TLinkAddress target)
346             {
347                 var root = GetTreeRoot();
348                 while (!EqualToZero(root))
349                 {
350                     ref var rootLink = ref GetLinkDataPartReference(root);
351                     var rootSource = rootLink.Source;
352                     var rootTarget = rootLink.Target;
353                     if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                        ↪ node.Key < root.Key
354                     {
355                         root = GetLeftOrDefault(root);
356                     }
357                     else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
                        ↪ node.Key > root.Key
358                     {
359                         root = GetRightOrDefault(root);
360                     }
361                     else // node.Key == root.Key
362                     {
363                         return root;
364                     }
365                 }
366                 return Zero;
367             }
368 
369             // TODO: Return indices range instead of references count
370             /// <summary>
371             /// <para>
372             /// Counts the usages using the specified link.
373             /// </para>
374             /// <para></para>
375             /// </summary>
376             /// <param name="link">
377             /// <para>The link.</para>
378             /// <para></para>
379             /// </param>
380             /// <returns>
381             /// <para>The link</para>
382             /// <para></para>
383             /// </returns>
384             [MethodImpl(MethodImplOptions.AggressiveInlining)]
385             public TLinkAddress CountUsages(TLinkAddress link)
386             {
387                 var root = GetTreeRoot();
388                 var total = GetSize(root);
389                 var totalRightIgnore = Zero;
390                 while (!EqualToZero(root))
391                 {
392                     var @base = GetBasePartValue(root);
393                     if (LessOrEqualThan(@base, link))
394                     {
395                         root = GetRightOrDefault(root);
396                     }
397                     else
398                     {
399                         totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
400                         root = GetLeftOrDefault(root);
401                     }
402                 }
403                 root = GetTreeRoot();
404                 var totalLeftIgnore = Zero;
405                 while (!EqualToZero(root))
406                 {
407                     var @base = GetBasePartValue(root);
408                     if (GreaterOrEqualThan(@base, link))
409                     {
410                         root = GetLeftOrDefault(root);
411                     }
412                     else
413                     {
414                         totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
415                         root = GetRightOrDefault(root);
416                     }
417                 }
418                 return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
```

```
419         }
420
421         /// <summary>
422         /// <para>
423         /// Eaches the usage using the specified base.
424         /// </para>
425         /// <para></para>
426         /// </summary>
427         /// <param name="@base">
428         /// <para>The base.</para>
429         /// <para></para>
430         /// </param>
431         /// <param name="handler">
432         /// <para>The handler.</para>
433         /// <para></para>
434         /// </param>
435         /// <returns>
436         /// <para>The link</para>
437         /// <para></para>
438         /// </returns>
439         [MethodImpl(MethodImplOptions.AggressiveInlining)]
440         public TLinkAddress EachUsage(TLinkAddress @base, ReadHandler<TLinkAddress>? handler) =>
            ↪ EachUsageCore(@base, GetTreeRoot(), handler);
441
442         // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
            ↪ low-level MSIL stack.
443         [MethodImpl(MethodImplOptions.AggressiveInlining)]
444         private TLinkAddress EachUsageCore(TLinkAddress @base, TLinkAddress link,
            ↪ ReadHandler<TLinkAddress>? handler)
445         {
446             var @continue = Continue;
447             if (EqualToZero(link))
448             {
449                 return @continue;
450             }
451             var linkBasePart = GetBasePartValue(link);
452             var @break = Break;
453             if (GreaterThan(linkBasePart, @base))
454             {
455                 if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
456                 {
457                     return @break;
458                 }
459             }
460             else if (LessThan(linkBasePart, @base))
461             {
462                 if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
463                 {
464                     return @break;
465                 }
466             }
467             else //if (linkBasePart == @base)
468             {
469                 if (AreEqual(handler(GetLinkValues(link)), @break))
470                 {
471                     return @break;
472                 }
473                 if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
474                 {
475                     return @break;
476                 }
477                 if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
478                 {
479                     return @break;
480                 }
481             }
482             return @continue;
483         }
484
485         /// <summary>
486         /// <para>
487         /// Prints the node value using the specified node.
488         /// </para>
489         /// <para></para>
490         /// </summary>
491         /// <param name="node">
492         /// <para>The node.</para>
493         /// <para></para>
```

```
494         /// </param>
495         /// <param name="sb">
496         /// <para>The sb.</para>
497         /// <para></para>
498         /// </param>
499         [MethodImpl(MethodImplOptions.AggressiveInlining)]
500         protected override void PrintNodeValue(TLinkAddress node, StringBuilder sb)
501         {
502             ref var link = ref GetLinkDataPartReference(node);
503             sb.Append(' ');
504             sb.Append(link.Source);
505             sb.Append('-');
506             sb.Append('>');
507             sb.Append(link.Target);
508         }
509     }
510 }
```

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

```
1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Memory.Split.Generic
6  {
7      /// <summary>
8      /// <para>
9      /// Represents the external links sources recursionless size balanced tree methods.
10     /// </para>
11     /// <para></para>
12     /// </summary>
13     /// <seealso cref="ExternalLinksRecursionlessSizeBalancedTreeMethodsBase{TLinkAddress}"/>
14     public unsafe class ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods<TLinkAddress> :
       ↪ ExternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLinkAddress>
15     {
16         /// <summary>
17         /// <para>
18         /// Initializes a new <see
           ↪ cref="ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods"/> instance.
19         /// </para>
20         /// <para></para>
21         /// </summary>
22         /// <param name="constants">
23         /// <para>A constants.</para>
24         /// <para></para>
25         /// </param>
26         /// <param name="linksDataParts">
27         /// <para>A links data parts.</para>
28         /// <para></para>
29         /// </param>
30         /// <param name="linksIndexParts">
31         /// <para>A links index parts.</para>
32         /// <para></para>
33         /// </param>
34         /// <param name="header">
35         /// <para>A header.</para>
36         /// <para></para>
37         /// </param>
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<TLinkAddr
           ↪ ess> constants, byte* linksDataParts, byte* linksIndexParts, byte* header) :
           ↪ base(constants, linksDataParts, linksIndexParts, header) { }
40
41         /// <summary>
42         /// <para>
43         /// Gets the left reference using the specified node.
44         /// </para>
45         /// <para></para>
46         /// </summary>
47         /// <param name="node">
48         /// <para>The node.</para>
49         /// <para></para>
50         /// </param>
51         /// <returns>
52         /// <para>The ref link</para>
53         /// <para></para>
54         /// </returns>
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
56          protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
            ↪ GetLinkIndexPartReference(node).LeftAsSource;
57
58          /// <summary>
59          /// <para>
60          /// Gets the right reference using the specified node.
61          /// </para>
62          /// <para></para>
63          /// </summary>
64          /// <param name="node">
65          /// <para>The node.</para>
66          /// <para></para>
67          /// </param>
68          /// <returns>
69          /// <para>The ref link</para>
70          /// <para></para>
71          /// </returns>
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
            ↪ GetLinkIndexPartReference(node).RightAsSource;
74
75          /// <summary>
76          /// <para>
77          /// Gets the left using the specified node.
78          /// </para>
79          /// <para></para>
80          /// </summary>
81          /// <param name="node">
82          /// <para>The node.</para>
83          /// <para></para>
84          /// </param>
85          /// <returns>
86          /// <para>The link</para>
87          /// <para></para>
88          /// </returns>
89          [MethodImpl(MethodImplOptions.AggressiveInlining)]
90          protected override TLinkAddress GetLeft(TLinkAddress node) =>
            ↪ GetLinkIndexPartReference(node).LeftAsSource;
91
92          /// <summary>
93          /// <para>
94          /// Gets the right using the specified node.
95          /// </para>
96          /// <para></para>
97          /// </summary>
98          /// <param name="node">
99          /// <para>The node.</para>
100         /// <para></para>
101         /// </param>
102         /// <returns>
103         /// <para>The link</para>
104         /// <para></para>
105         /// </returns>
106         [MethodImpl(MethodImplOptions.AggressiveInlining)]
107         protected override TLinkAddress GetRight(TLinkAddress node) =>
            ↪ GetLinkIndexPartReference(node).RightAsSource;
108
109         /// <summary>
110         /// <para>
111         /// Sets the left using the specified node.
112         /// </para>
113         /// <para></para>
114         /// </summary>
115         /// <param name="node">
116         /// <para>The node.</para>
117         /// <para></para>
118         /// </param>
119         /// <param name="left">
120         /// <para>The left.</para>
121         /// <para></para>
122         /// </param>
123         [MethodImpl(MethodImplOptions.AggressiveInlining)]
124         protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
            ↪ GetLinkIndexPartReference(node).LeftAsSource = left;
125
126         /// <summary>
127         /// <para>
```

```
128             /// Sets the right using the specified node.
129             /// </para>
130             /// <para></para>
131             /// </summary>
132             /// <param name="node">
133             /// <para>The node.</para>
134             /// <para></para>
135             /// </param>
136             /// <param name="right">
137             /// <para>The right.</para>
138             /// <para></para>
139             /// </param>
140             [MethodImpl(MethodImplOptions.AggressiveInlining)]
141             protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
                ↪ GetLinkIndexPartReference(node).RightAsSource = right;
142
143             /// <summary>
144             /// <para>
145             /// Gets the size using the specified node.
146             /// </para>
147             /// <para></para>
148             /// </summary>
149             /// <param name="node">
150             /// <para>The node.</para>
151             /// <para></para>
152             /// </param>
153             /// <returns>
154             /// <para>The link</para>
155             /// <para></para>
156             /// </returns>
157             [MethodImpl(MethodImplOptions.AggressiveInlining)]
158             protected override TLinkAddress GetSize(TLinkAddress node) =>
                ↪ GetLinkIndexPartReference(node).SizeAsSource;
159
160             /// <summary>
161             /// <para>
162             /// Sets the size using the specified node.
163             /// </para>
164             /// <para></para>
165             /// </summary>
166             /// <param name="node">
167             /// <para>The node.</para>
168             /// <para></para>
169             /// </param>
170             /// <param name="size">
171             /// <para>The size.</para>
172             /// <para></para>
173             /// </param>
174             [MethodImpl(MethodImplOptions.AggressiveInlining)]
175             protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
                ↪ GetLinkIndexPartReference(node).SizeAsSource = size;
176
177             /// <summary>
178             /// <para>
179             /// Gets the tree root.
180             /// </para>
181             /// <para></para>
182             /// </summary>
183             /// <returns>
184             /// <para>The link</para>
185             /// <para></para>
186             /// </returns>
187             [MethodImpl(MethodImplOptions.AggressiveInlining)]
188             protected override TLinkAddress GetTreeRoot() => GetHeaderReference().RootAsSource;
189
190             /// <summary>
191             /// <para>
192             /// Gets the base part value using the specified link.
193             /// </para>
194             /// <para></para>
195             /// </summary>
196             /// <param name="link">
197             /// <para>The link.</para>
198             /// <para></para>
199             /// </param>
200             /// <returns>
201             /// <para>The link</para>
202             /// <para></para>
```

```
203             /// </returns>
204             [MethodImpl(MethodImplOptions.AggressiveInlining)]
205             protected override TLinkAddress GetBasePartValue(TLinkAddress link) =>
                ↪ GetLinkDataPartReference(link).Source;
206
207             /// <summary>
208             /// <para>
209             /// Determines whether this instance first is to the left of second.
210             /// </para>
211             /// <para></para>
212             /// </summary>
213             /// <param name="firstSource">
214             /// <para>The first source.</para>
215             /// <para></para>
216             /// </param>
217             /// <param name="firstTarget">
218             /// <para>The first target.</para>
219             /// <para></para>
220             /// </param>
221             /// <param name="secondSource">
222             /// <para>The second source.</para>
223             /// <para></para>
224             /// </param>
225             /// <param name="secondTarget">
226             /// <para>The second target.</para>
227             /// <para></para>
228             /// </param>
229             /// <returns>
230             /// <para>The bool</para>
231             /// <para></para>
232             /// </returns>
233             [MethodImpl(MethodImplOptions.AggressiveInlining)]
234             protected override bool FirstIsToTheLeftOfSecond(TLinkAddress firstSource, TLinkAddress
                ↪ firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget) =>
                ↪ LessThan(firstSource, secondSource) || (AreEqual(firstSource, secondSource) &&
                ↪ LessThan(firstTarget, secondTarget));
235
236             /// <summary>
237             /// <para>
238             /// Determines whether this instance first is to the right of second.
239             /// </para>
240             /// <para></para>
241             /// </summary>
242             /// <param name="firstSource">
243             /// <para>The first source.</para>
244             /// <para></para>
245             /// </param>
246             /// <param name="firstTarget">
247             /// <para>The first target.</para>
248             /// <para></para>
249             /// </param>
250             /// <param name="secondSource">
251             /// <para>The second source.</para>
252             /// <para></para>
253             /// </param>
254             /// <param name="secondTarget">
255             /// <para>The second target.</para>
256             /// <para></para>
257             /// </param>
258             /// <returns>
259             /// <para>The bool</para>
260             /// <para></para>
261             /// </returns>
262             [MethodImpl(MethodImplOptions.AggressiveInlining)]
263             protected override bool FirstIsToTheRightOfSecond(TLinkAddress firstSource, TLinkAddress
                ↪ firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget) =>
                ↪ GreaterThan(firstSource, secondSource) || (AreEqual(firstSource, secondSource) &&
                ↪ GreaterThan(firstTarget, secondTarget));
264
265             /// <summary>
266             /// <para>
267             /// Clears the node using the specified node.
268             /// </para>
269             /// <para></para>
270             /// </summary>
271             /// <param name="node">
272             /// <para>The node.</para>
```

```
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLinkAddress node)
        {
            ref var link = ref GetLinkIndexPartReference(node);
            link.LeftAsSource = Zero;
            link.RightAsSource = Zero;
            link.SizeAsSource = Zero;
        }
    }
}
```

1.35 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/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
{
    /// <summary>
    /// <para>
    /// Represents the external links sources size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ExternalLinksSizeBalancedTreeMethodsBase{TLinkAddress}"/>
    public unsafe class ExternalLinksSourcesSizeBalancedTreeMethods<TLinkAddress> :
    ↪ ExternalLinksSizeBalancedTreeMethodsBase<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="ExternalLinksSourcesSizeBalancedTreeMethods"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksDataParts">
        /// <para>A links data parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksIndexParts">
        /// <para>A links index parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ExternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLinkAddress>
        ↪ constants, byte* linksDataParts, byte* linksIndexParts, byte* header) :
        ↪ base(constants, linksDataParts, linksIndexParts, header) { }

        /// <summary>
        /// <para>
        /// Gets the left reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
        ↪ GetLinkIndexPartReference(node).LeftAsSource;

        /// <summary>
        /// <para>
        /// Gets the right reference using the specified node.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
        ↪ GetLinkIndexPartReference(node).RightAsSource;

        /// <summary>
        /// <para>
        /// Gets the left using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetLeft(TLinkAddress node) =>
        ↪ GetLinkIndexPartReference(node).LeftAsSource;

        /// <summary>
        /// <para>
        /// Gets the right using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetRight(TLinkAddress node) =>
        ↪ GetLinkIndexPartReference(node).RightAsSource;

        /// <summary>
        /// <para>
        /// Sets the left using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="left">
        /// <para>The left.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
        ↪ GetLinkIndexPartReference(node).LeftAsSource = left;

        /// <summary>
        /// <para>
        /// Sets the right using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
```

```
135             /// </param>
136             /// <param name="right">
137             /// <para>The right.</para>
138             /// <para></para>
139             /// </param>
140             [MethodImpl(MethodImplOptions.AggressiveInlining)]
141             protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
                ↪ GetLinkIndexPartReference(node).RightAsSource = right;
142
143             /// <summary>
144             /// <para>
145             /// Gets the size using the specified node.
146             /// </para>
147             /// <para></para>
148             /// </summary>
149             /// <param name="node">
150             /// <para>The node.</para>
151             /// <para></para>
152             /// </param>
153             /// <returns>
154             /// <para>The link</para>
155             /// <para></para>
156             /// </returns>
157             [MethodImpl(MethodImplOptions.AggressiveInlining)]
158             protected override TLinkAddress GetSize(TLinkAddress node) =>
                ↪ GetLinkIndexPartReference(node).SizeAsSource;
159
160             /// <summary>
161             /// <para>
162             /// Sets the size using the specified node.
163             /// </para>
164             /// <para></para>
165             /// </summary>
166             /// <param name="node">
167             /// <para>The node.</para>
168             /// <para></para>
169             /// </param>
170             /// <param name="size">
171             /// <para>The size.</para>
172             /// <para></para>
173             /// </param>
174             [MethodImpl(MethodImplOptions.AggressiveInlining)]
175             protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
                ↪ GetLinkIndexPartReference(node).SizeAsSource = size;
176
177             /// <summary>
178             /// <para>
179             /// Gets the tree root.
180             /// </para>
181             /// <para></para>
182             /// </summary>
183             /// <returns>
184             /// <para>The link</para>
185             /// <para></para>
186             /// </returns>
187             [MethodImpl(MethodImplOptions.AggressiveInlining)]
188             protected override TLinkAddress GetTreeRoot() => GetHeaderReference().RootAsSource;
189
190             /// <summary>
191             /// <para>
192             /// Gets the base part value using the specified link.
193             /// </para>
194             /// <para></para>
195             /// </summary>
196             /// <param name="link">
197             /// <para>The link.</para>
198             /// <para></para>
199             /// </param>
200             /// <returns>
201             /// <para>The link</para>
202             /// <para></para>
203             /// </returns>
204             [MethodImpl(MethodImplOptions.AggressiveInlining)]
205             protected override TLinkAddress GetBasePartValue(TLinkAddress link) =>
                ↪ GetLinkDataPartReference(link).Source;
206
207             /// <summary>
208             /// <para>
```

```
209         /// Determines whether this instance first is to the left of second.
210         /// </para>
211         /// <para></para>
212         /// </summary>
213         /// <param name="firstSource">
214         /// <para>The first source.</para>
215         /// <para></para>
216         /// </param>
217         /// <param name="firstTarget">
218         /// <para>The first target.</para>
219         /// <para></para>
220         /// </param>
221         /// <param name="secondSource">
222         /// <para>The second source.</para>
223         /// <para></para>
224         /// </param>
225         /// <param name="secondTarget">
226         /// <para>The second target.</para>
227         /// <para></para>
228         /// </param>
229         /// <returns>
230         /// <para>The bool</para>
231         /// <para></para>
232         /// </returns>
233         [MethodImpl(MethodImplOptions.AggressiveInlining)]
234         protected override bool FirstIsToTheLeftOfSecond(TLinkAddress firstSource, TLinkAddress
              ↪ firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget) =>
              ↪ LessThan(firstSource, secondSource) || (AreEqual(firstSource, secondSource) &&
              ↪ LessThan(firstTarget, secondTarget));
235
236         /// <summary>
237         /// <para>
238         /// Determines whether this instance first is to the right of second.
239         /// </para>
240         /// <para></para>
241         /// </summary>
242         /// <param name="firstSource">
243         /// <para>The first source.</para>
244         /// <para></para>
245         /// </param>
246         /// <param name="firstTarget">
247         /// <para>The first target.</para>
248         /// <para></para>
249         /// </param>
250         /// <param name="secondSource">
251         /// <para>The second source.</para>
252         /// <para></para>
253         /// </param>
254         /// <param name="secondTarget">
255         /// <para>The second target.</para>
256         /// <para></para>
257         /// </param>
258         /// <returns>
259         /// <para>The bool</para>
260         /// <para></para>
261         /// </returns>
262         [MethodImpl(MethodImplOptions.AggressiveInlining)]
263         protected override bool FirstIsToTheRightOfSecond(TLinkAddress firstSource, TLinkAddress
              ↪ firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget) =>
              ↪ GreaterThan(firstSource, secondSource) || (AreEqual(firstSource, secondSource) &&
              ↪ GreaterThan(firstTarget, secondTarget));
264
265         /// <summary>
266         /// <para>
267         /// Clears the node using the specified node.
268         /// </para>
269         /// <para></para>
270         /// </summary>
271         /// <param name="node">
272         /// <para>The node.</para>
273         /// <para></para>
274         /// </param>
275         [MethodImpl(MethodImplOptions.AggressiveInlining)]
276         protected override void ClearNode(TLinkAddress node)
277         {
278             ref var link = ref GetLinkIndexPartReference(node);
279             link.LeftAsSource = Zero;
```

```
            link.RightAsSource = Zero;
            link.SizeAsSource = Zero;
        }
    }
}
```

1.36 ./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
{
    /// <summary>
    /// <para>
    /// Represents the external links targets recursionless size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ExternalLinksRecursionlessSizeBalancedTreeMethodsBase{TLinkAddress}"/>
    public unsafe class ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods<TLinkAddress> :
    ↪ ExternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see
        ↪ cref="ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksDataParts">
        /// <para>A links data parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksIndexParts">
        /// <para>A links index parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<TLinkAddr
        ↪ ess> constants, byte* linksDataParts, byte* linksIndexParts, byte* header) :
        ↪ base(constants, linksDataParts, linksIndexParts, header) { }

        /// <summary>
        /// <para>
        /// Gets the left reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
        ↪ GetLinkIndexPartReference(node).LeftAsTarget;

        /// <summary>
        /// <para>
        /// Gets the right reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
```

```
67          /// </param>
68          /// <returns>
69          /// <para>The ref link</para>
70          /// <para></para>
71          /// </returns>
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
            ↪ GetLinkIndexPartReference(node).RightAsTarget;
74
75          /// <summary>
76          /// <para>
77          /// Gets the left using the specified node.
78          /// </para>
79          /// <para></para>
80          /// </summary>
81          /// <param name="node">
82          /// <para>The node.</para>
83          /// <para></para>
84          /// </param>
85          /// <returns>
86          /// <para>The link</para>
87          /// <para></para>
88          /// </returns>
89          [MethodImpl(MethodImplOptions.AggressiveInlining)]
90          protected override TLinkAddress GetLeft(TLinkAddress node) =>
            ↪ GetLinkIndexPartReference(node).LeftAsTarget;
91
92          /// <summary>
93          /// <para>
94          /// Gets the right using the specified node.
95          /// </para>
96          /// <para></para>
97          /// </summary>
98          /// <param name="node">
99          /// <para>The node.</para>
100         /// <para></para>
101         /// </param>
102         /// <returns>
103         /// <para>The link</para>
104         /// <para></para>
105         /// </returns>
106         [MethodImpl(MethodImplOptions.AggressiveInlining)]
107         protected override TLinkAddress GetRight(TLinkAddress node) =>
            ↪ GetLinkIndexPartReference(node).RightAsTarget;
108
109         /// <summary>
110         /// <para>
111         /// Sets the left using the specified node.
112         /// </para>
113         /// <para></para>
114         /// </summary>
115         /// <param name="node">
116         /// <para>The node.</para>
117         /// <para></para>
118         /// </param>
119         /// <param name="left">
120         /// <para>The left.</para>
121         /// <para></para>
122         /// </param>
123         [MethodImpl(MethodImplOptions.AggressiveInlining)]
124         protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
            ↪ GetLinkIndexPartReference(node).LeftAsTarget = left;
125
126         /// <summary>
127         /// <para>
128         /// Sets the right using the specified node.
129         /// </para>
130         /// <para></para>
131         /// </summary>
132         /// <param name="node">
133         /// <para>The node.</para>
134         /// <para></para>
135         /// </param>
136         /// <param name="right">
137         /// <para>The right.</para>
138         /// <para></para>
139         /// </param>
140         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
141         protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
            ↪ GetLinkIndexPartReference(node).RightAsTarget = right;
142
143         /// <summary>
144         /// <para>
145         /// Gets the size using the specified node.
146         /// </para>
147         /// <para></para>
148         /// </summary>
149         /// <param name="node">
150         /// <para>The node.</para>
151         /// <para></para>
152         /// </param>
153         /// <returns>
154         /// <para>The link</para>
155         /// <para></para>
156         /// </returns>
157         [MethodImpl(MethodImplOptions.AggressiveInlining)]
158         protected override TLinkAddress GetSize(TLinkAddress node) =>
            ↪ GetLinkIndexPartReference(node).SizeAsTarget;
159
160         /// <summary>
161         /// <para>
162         /// Sets the size using the specified node.
163         /// </para>
164         /// <para></para>
165         /// </summary>
166         /// <param name="node">
167         /// <para>The node.</para>
168         /// <para></para>
169         /// </param>
170         /// <param name="size">
171         /// <para>The size.</para>
172         /// <para></para>
173         /// </param>
174         [MethodImpl(MethodImplOptions.AggressiveInlining)]
175         protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
            ↪ GetLinkIndexPartReference(node).SizeAsTarget = size;
176
177         /// <summary>
178         /// <para>
179         /// Gets the tree root.
180         /// </para>
181         /// <para></para>
182         /// </summary>
183         /// <returns>
184         /// <para>The link</para>
185         /// <para></para>
186         /// </returns>
187         [MethodImpl(MethodImplOptions.AggressiveInlining)]
188         protected override TLinkAddress GetTreeRoot() => GetHeaderReference().RootAsTarget;
189
190         /// <summary>
191         /// <para>
192         /// Gets the base part value using the specified link.
193         /// </para>
194         /// <para></para>
195         /// </summary>
196         /// <param name="link">
197         /// <para>The link.</para>
198         /// <para></para>
199         /// </param>
200         /// <returns>
201         /// <para>The link</para>
202         /// <para></para>
203         /// </returns>
204         [MethodImpl(MethodImplOptions.AggressiveInlining)]
205         protected override TLinkAddress GetBasePartValue(TLinkAddress link) =>
            ↪ GetLinkDataPartReference(link).Target;
206
207         /// <summary>
208         /// <para>
209         /// Determines whether this instance first is to the left of second.
210         /// </para>
211         /// <para></para>
212         /// </summary>
213         /// <param name="firstSource">
214         /// <para>The first source.</para>
```

```
215             /// <para></para>
216             /// </param>
217             /// <param name="firstTarget">
218             /// <para>The first target.</para>
219             /// <para></para>
220             /// </param>
221             /// <param name="secondSource">
222             /// <para>The second source.</para>
223             /// <para></para>
224             /// </param>
225             /// <param name="secondTarget">
226             /// <para>The second target.</para>
227             /// <para></para>
228             /// </param>
229             /// <returns>
230             /// <para>The bool</para>
231             /// <para></para>
232             /// </returns>
233             [MethodImpl(MethodImplOptions.AggressiveInlining)]
234             protected override bool FirstIsToTheLeftOfSecond(TLinkAddress firstSource, TLinkAddress
                ↪ firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget) =>
                ↪ LessThan(firstTarget, secondTarget) || (AreEqual(firstTarget, secondTarget) &&
                ↪ LessThan(firstSource, secondSource));
235
236             /// <summary>
237             /// <para>
238             /// Determines whether this instance first is to the right of second.
239             /// </para>
240             /// <para></para>
241             /// </summary>
242             /// <param name="firstSource">
243             /// <para>The first source.</para>
244             /// <para></para>
245             /// </param>
246             /// <param name="firstTarget">
247             /// <para>The first target.</para>
248             /// <para></para>
249             /// </param>
250             /// <param name="secondSource">
251             /// <para>The second source.</para>
252             /// <para></para>
253             /// </param>
254             /// <param name="secondTarget">
255             /// <para>The second target.</para>
256             /// <para></para>
257             /// </param>
258             /// <returns>
259             /// <para>The bool</para>
260             /// <para></para>
261             /// </returns>
262             [MethodImpl(MethodImplOptions.AggressiveInlining)]
263             protected override bool FirstIsToTheRightOfSecond(TLinkAddress firstSource, TLinkAddress
                ↪ firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget) =>
                ↪ GreaterThan(firstTarget, secondTarget) || (AreEqual(firstTarget, secondTarget) &&
                ↪ GreaterThan(firstSource, secondSource));
264
265             /// <summary>
266             /// <para>
267             /// Clears the node using the specified node.
268             /// </para>
269             /// <para></para>
270             /// </summary>
271             /// <param name="node">
272             /// <para>The node.</para>
273             /// <para></para>
274             /// </param>
275             [MethodImpl(MethodImplOptions.AggressiveInlining)]
276             protected override void ClearNode(TLinkAddress node)
277             {
278                 ref var link = ref GetLinkIndexPartReference(node);
279                 link.LeftAsTarget = Zero;
280                 link.RightAsTarget = Zero;
281                 link.SizeAsTarget = Zero;
282             }
283         }
284     }
```

```
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Memory.Split.Generic
{
    /// <summary>
    /// <para>
    /// Represents the external links targets size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ExternalLinksSizeBalancedTreeMethodsBase{TLinkAddress}"/>
    public unsafe class ExternalLinksTargetsSizeBalancedTreeMethods<TLinkAddress> :
    ↪  ExternalLinksSizeBalancedTreeMethodsBase<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="ExternalLinksTargetsSizeBalancedTreeMethods"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksDataParts">
        /// <para>A links data parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksIndexParts">
        /// <para>A links index parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ExternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLinkAddress>
        ↪  constants, byte* linksDataParts, byte* linksIndexParts, byte* header) :
        ↪  base(constants, linksDataParts, linksIndexParts, header) { }

        /// <summary>
        /// <para>
        /// Gets the left reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
        ↪  GetLinkIndexPartReference(node).LeftAsTarget;

        /// <summary>
        /// <para>
        /// Gets the right reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
73          protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
            ↪ GetLinkIndexPartReference(node).RightAsTarget;
74
75          /// <summary>
76          /// <para>
77          /// Gets the left using the specified node.
78          /// </para>
79          /// <para></para>
80          /// </summary>
81          /// <param name="node">
82          /// <para>The node.</para>
83          /// <para></para>
84          /// </param>
85          /// <returns>
86          /// <para>The link</para>
87          /// <para></para>
88          /// </returns>
89          [MethodImpl(MethodImplOptions.AggressiveInlining)]
90          protected override TLinkAddress GetLeft(TLinkAddress node) =>
            ↪ GetLinkIndexPartReference(node).LeftAsTarget;
91
92          /// <summary>
93          /// <para>
94          /// Gets the right using the specified node.
95          /// </para>
96          /// <para></para>
97          /// </summary>
98          /// <param name="node">
99          /// <para>The node.</para>
100         /// <para></para>
101         /// </param>
102         /// <returns>
103         /// <para>The link</para>
104         /// <para></para>
105         /// </returns>
106         [MethodImpl(MethodImplOptions.AggressiveInlining)]
107         protected override TLinkAddress GetRight(TLinkAddress node) =>
            ↪ GetLinkIndexPartReference(node).RightAsTarget;
108
109         /// <summary>
110         /// <para>
111         /// Sets the left using the specified node.
112         /// </para>
113         /// <para></para>
114         /// </summary>
115         /// <param name="node">
116         /// <para>The node.</para>
117         /// <para></para>
118         /// </param>
119         /// <param name="left">
120         /// <para>The left.</para>
121         /// <para></para>
122         /// </param>
123         [MethodImpl(MethodImplOptions.AggressiveInlining)]
124         protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
            ↪ GetLinkIndexPartReference(node).LeftAsTarget = left;
125
126         /// <summary>
127         /// <para>
128         /// Sets the right using the specified node.
129         /// </para>
130         /// <para></para>
131         /// </summary>
132         /// <param name="node">
133         /// <para>The node.</para>
134         /// <para></para>
135         /// </param>
136         /// <param name="right">
137         /// <para>The right.</para>
138         /// <para></para>
139         /// </param>
140         [MethodImpl(MethodImplOptions.AggressiveInlining)]
141         protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
            ↪ GetLinkIndexPartReference(node).RightAsTarget = right;
142
143         /// <summary>
144         /// <para>
```

```
        /// Gets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetSize(TLinkAddress node) =>
        ↪  GetLinkIndexPartReference(node).SizeAsTarget;

        /// <summary>
        /// <para>
        /// Sets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="size">
        /// <para>The size.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
        ↪  GetLinkIndexPartReference(node).SizeAsTarget = size;

        /// <summary>
        /// <para>
        /// Gets the tree root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetTreeRoot() => GetHeaderReference().RootAsTarget;

        /// <summary>
        /// <para>
        /// Gets the base part value using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetBasePartValue(TLinkAddress link) =>
        ↪  GetLinkDataPartReference(link).Target;

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="firstSource">
        /// <para>The first source.</para>
        /// <para></para>
        /// </param>
        /// <param name="firstTarget">
        /// <para>The first target.</para>
        /// <para></para>
```

```
220             /// </param>
221             /// <param name="secondSource">
222             /// <para>The second source.</para>
223             /// <para></para>
224             /// </param>
225             /// <param name="secondTarget">
226             /// <para>The second target.</para>
227             /// <para></para>
228             /// </param>
229             /// <returns>
230             /// <para>The bool</para>
231             /// <para></para>
232             /// </returns>
233             [MethodImpl(MethodImplOptions.AggressiveInlining)]
234             protected override bool FirstIsToTheLeftOfSecond(TLinkAddress firstSource, TLinkAddress
                ↪ firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget) =>
                ↪ LessThan(firstTarget, secondTarget) || (AreEqual(firstTarget, secondTarget) &&
                ↪ LessThan(firstSource, secondSource));
235
236             /// <summary>
237             /// <para>
238             /// Determines whether this instance first is to the right of second.
239             /// </para>
240             /// <para></para>
241             /// </summary>
242             /// <param name="firstSource">
243             /// <para>The first source.</para>
244             /// <para></para>
245             /// </param>
246             /// <param name="firstTarget">
247             /// <para>The first target.</para>
248             /// <para></para>
249             /// </param>
250             /// <param name="secondSource">
251             /// <para>The second source.</para>
252             /// <para></para>
253             /// </param>
254             /// <param name="secondTarget">
255             /// <para>The second target.</para>
256             /// <para></para>
257             /// </param>
258             /// <returns>
259             /// <para>The bool</para>
260             /// <para></para>
261             /// </returns>
262             [MethodImpl(MethodImplOptions.AggressiveInlining)]
263             protected override bool FirstIsToTheRightOfSecond(TLinkAddress firstSource, TLinkAddress
                ↪ firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget) =>
                ↪ GreaterThan(firstTarget, secondTarget) || (AreEqual(firstTarget, secondTarget) &&
                ↪ GreaterThan(firstSource, secondSource));
264
265             /// <summary>
266             /// <para>
267             /// Clears the node using the specified node.
268             /// </para>
269             /// <para></para>
270             /// </summary>
271             /// <param name="node">
272             /// <para>The node.</para>
273             /// <para></para>
274             /// </param>
275             [MethodImpl(MethodImplOptions.AggressiveInlining)]
276             protected override void ClearNode(TLinkAddress node)
277             {
278                 ref var link = ref GetLinkIndexPartReference(node);
279                 link.LeftAsTarget = Zero;
280                 link.RightAsTarget = Zero;
281                 link.SizeAsTarget = Zero;
282             }
283         }
284     }
```

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

```
1   using System;
2   using System.Text;
3   using System.Collections.Generic;
4   using System.Runtime.CompilerServices;
5   using Platform.Collections.Methods.Trees;
```

```
using Platform.Converters;
using Platform.Delegates;
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
{
    /// <summary>
    /// <para>
    /// Represents the internal links recursionless size balanced tree methods base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="RecursionlessSizeBalancedTreeMethods{TLinkAddress}"/>
    /// <seealso cref="ILinksTreeMethods{TLinkAddress}"/>
    public unsafe abstract class InternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLinkAddress> : RecursionlessSizeBalancedTreeMethods<TLinkAddress>, ILinksTreeMethods<TLinkAddress>
    {
        private static readonly UncheckedConverter<TLinkAddress, long> _addressToInt64Converter = UncheckedConverter<TLinkAddress, long>.Default;

        /// <summary>
        /// <para>
        /// The break.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly TLinkAddress Break;
        /// <summary>
        /// <para>
        /// The continue.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly TLinkAddress Continue;
        /// <summary>
        /// <para>
        /// The links data parts.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly byte* LinksDataParts;
        /// <summary>
        /// <para>
        /// The links index parts.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly byte* LinksIndexParts;
        /// <summary>
        /// <para>
        /// The header.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly byte* Header;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="InternalLinksRecursionlessSizeBalancedTreeMethodsBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksDataParts">
        /// <para>A links data parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksIndexParts">
        /// <para>A links index parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
```

```
81          /// <para>A header.</para>
82          /// <para></para>
83          /// </param>
84          [MethodImpl(MethodImplOptions.AggressiveInlining)]
85          protected
            ↪  InternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLinkAddress>
            ↪  constants, byte* linksDataParts, byte* linksIndexParts, byte* header)
86          {
87              LinksDataParts = linksDataParts;
88              LinksIndexParts = linksIndexParts;
89              Header = header;
90              Break = constants.Break;
91              Continue = constants.Continue;
92          }
93
94          /// <summary>
95          /// <para>
96          /// Gets the tree root using the specified link.
97          /// </para>
98          /// <para></para>
99          /// </summary>
100         /// <param name="link">
101         /// <para>The link.</para>
102         /// <para></para>
103         /// </param>
104         /// <returns>
105         /// <para>The link</para>
106         /// <para></para>
107         /// </returns>
108         [MethodImpl(MethodImplOptions.AggressiveInlining)]
109         protected abstract TLinkAddress GetTreeRoot(TLinkAddress link);
110
111         /// <summary>
112         /// <para>
113         /// Gets the base part value using the specified link.
114         /// </para>
115         /// <para></para>
116         /// </summary>
117         /// <param name="link">
118         /// <para>The link.</para>
119         /// <para></para>
120         /// </param>
121         /// <returns>
122         /// <para>The link</para>
123         /// <para></para>
124         /// </returns>
125         [MethodImpl(MethodImplOptions.AggressiveInlining)]
126         protected abstract TLinkAddress GetBasePartValue(TLinkAddress link);
127
128         /// <summary>
129         /// <para>
130         /// Gets the key part value using the specified link.
131         /// </para>
132         /// <para></para>
133         /// </summary>
134         /// <param name="link">
135         /// <para>The link.</para>
136         /// <para></para>
137         /// </param>
138         /// <returns>
139         /// <para>The link</para>
140         /// <para></para>
141         /// </returns>
142         [MethodImpl(MethodImplOptions.AggressiveInlining)]
143         protected abstract TLinkAddress GetKeyPartValue(TLinkAddress link);
144
145         /// <summary>
146         /// <para>
147         /// Gets the link data part reference using the specified link.
148         /// </para>
149         /// <para></para>
150         /// </summary>
151         /// <param name="link">
152         /// <para>The link.</para>
153         /// <para></para>
154         /// </param>
155         /// <returns>
156         /// <para>A ref raw link data part of t link</para>
```

```
157             /// <para></para>
158             /// </returns>
159             [MethodImpl(MethodImplOptions.AggressiveInlining)]
160             protected virtual ref RawLinkDataPart<TLinkAddress>
                ↪  GetLinkDataPartReference(TLinkAddress link) => ref
                ↪  AsRef<RawLinkDataPart<TLinkAddress>>(LinksDataParts +
                ↪  (RawLinkDataPart<TLinkAddress>.SizeInBytes *
                ↪  _addressToInt64Converter.Convert(link)));
161
162             /// <summary>
163             /// <para>
164             /// Gets the link index part reference using the specified link.
165             /// </para>
166             /// <para></para>
167             /// </summary>
168             /// <param name="link">
169             /// <para>The link.</para>
170             /// <para></para>
171             /// </param>
172             /// <returns>
173             /// <para>A ref raw link index part of t link</para>
174             /// <para></para>
175             /// </returns>
176             [MethodImpl(MethodImplOptions.AggressiveInlining)]
177             protected virtual ref RawLinkIndexPart<TLinkAddress>
                ↪  GetLinkIndexPartReference(TLinkAddress link) => ref
                ↪  AsRef<RawLinkIndexPart<TLinkAddress>>(LinksIndexParts +
                ↪  (RawLinkIndexPart<TLinkAddress>.SizeInBytes *
                ↪  _addressToInt64Converter.Convert(link)));
178
179             /// <summary>
180             /// <para>
181             /// Determines whether this instance first is to the left of second.
182             /// </para>
183             /// <para></para>
184             /// </summary>
185             /// <param name="first">
186             /// <para>The first.</para>
187             /// <para></para>
188             /// </param>
189             /// <param name="second">
190             /// <para>The second.</para>
191             /// <para></para>
192             /// </param>
193             /// <returns>
194             /// <para>The bool</para>
195             /// <para></para>
196             /// </returns>
197             [MethodImpl(MethodImplOptions.AggressiveInlining)]
198             protected override bool FirstIsToTheLeftOfSecond(TLinkAddress first, TLinkAddress
                ↪  second) => LessThan(GetKeyPartValue(first), GetKeyPartValue(second));
199
200             /// <summary>
201             /// <para>
202             /// Determines whether this instance first is to the right of second.
203             /// </para>
204             /// <para></para>
205             /// </summary>
206             /// <param name="first">
207             /// <para>The first.</para>
208             /// <para></para>
209             /// </param>
210             /// <param name="second">
211             /// <para>The second.</para>
212             /// <para></para>
213             /// </param>
214             /// <returns>
215             /// <para>The bool</para>
216             /// <para></para>
217             /// </returns>
218             [MethodImpl(MethodImplOptions.AggressiveInlining)]
219             protected override bool FirstIsToTheRightOfSecond(TLinkAddress first, TLinkAddress
                ↪  second) => GreaterThan(GetKeyPartValue(first), GetKeyPartValue(second));
220
221             /// <summary>
222             /// <para>
223             /// Gets the link values using the specified link index.
224             /// </para>
```

```
        /// <para></para>
        /// </summary>
        /// <param name="linkIndex">
        /// <para>The link index.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A list of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual IList<TLinkAddress>? GetLinkValues(TLinkAddress linkIndex)
        {
            ref var link = ref GetLinkDataPartReference(linkIndex);
            return new Link<TLinkAddress>(linkIndex, link.Source, link.Target);
        }

        /// <summary>
        /// <para>
        /// The zero.
        /// </para>
        /// <para></para>
        /// </summary>
        public TLinkAddress this[TLinkAddress link, TLinkAddress 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 TLinkAddress Search(TLinkAddress source, TLinkAddress target);

        /// <summary>
        /// <para>
        /// Searches the core using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="key">
        /// <para>The key.</para>
        /// <para></para>
        /// </param>
```

```
        /// <returns>
        /// <para>The zero.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected TLinkAddress SearchCore(TLinkAddress root, TLinkAddress 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
        /// <summary>
        /// <para>
        /// Counts the usages using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress CountUsages(TLinkAddress link) => GetSizeOrZero(GetTreeRoot(link));

        /// <summary>
        /// <para>
        /// Eaches the usage using the specified base.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="@base">
        /// <para>The base.</para>
        /// <para></para>
        /// </param>
        /// <param name="handler">
        /// <para>The handler.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress EachUsage(TLinkAddress @base, ReadHandler<TLinkAddress>? 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 TLinkAddress EachUsageCore(TLinkAddress @base, TLinkAddress link,
        ↪  ReadHandler<TLinkAddress>? 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;
        }

        /// <summary>
        /// <para>
        /// Prints the node value using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="sb">
        /// <para>The sb.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void PrintNodeValue(TLinkAddress node, StringBuilder sb)
        {
            ref var link = ref GetLinkDataPartReference(node);
            sb.Append(' ');
            sb.Append(link.Source);
            sb.Append('-');
            sb.Append('>');
            sb.Append(link.Target);
        }
    }
}
```

1.39 ./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 Platform.Delegates;
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
{
    /// <summary>
    /// <para>
    /// Represents the internal links size balanced tree methods base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="SizeBalancedTreeMethods{TLinkAddress}"/>
    /// <seealso cref="ILinksTreeMethods{TLinkAddress}"/>
    public unsafe abstract class InternalLinksSizeBalancedTreeMethodsBase<TLinkAddress> :
    ↪  SizeBalancedTreeMethods<TLinkAddress>, ILinksTreeMethods<TLinkAddress>
    {
        private static readonly UncheckedConverter<TLinkAddress, long> _addressToInt64Converter
        ↪  = UncheckedConverter<TLinkAddress, long>.Default;

        /// <summary>
        /// <para>
        /// The break.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly TLinkAddress Break;
        /// <summary>
        /// <para>
```

```
35          /// The continue.
36          /// </para>
37          /// <para></para>
38          /// </summary>
39          protected readonly TLinkAddress Continue;
40          /// <summary>
41          /// <para>
42          /// The links data parts.
43          /// </para>
44          /// <para></para>
45          /// </summary>
46          protected readonly byte* LinksDataParts;
47          /// <summary>
48          /// <para>
49          /// The links index parts.
50          /// </para>
51          /// <para></para>
52          /// </summary>
53          protected readonly byte* LinksIndexParts;
54          /// <summary>
55          /// <para>
56          /// The header.
57          /// </para>
58          /// <para></para>
59          /// </summary>
60          protected readonly byte* Header;
61
62          /// <summary>
63          /// <para>
64          /// Initializes a new <see cref="InternalLinksSizeBalancedTreeMethodsBase"/> instance.
65          /// </para>
66          /// <para></para>
67          /// </summary>
68          /// <param name="constants">
69          /// <para>A constants.</para>
70          /// <para></para>
71          /// </param>
72          /// <param name="linksDataParts">
73          /// <para>A links data parts.</para>
74          /// <para></para>
75          /// </param>
76          /// <param name="linksIndexParts">
77          /// <para>A links index parts.</para>
78          /// <para></para>
79          /// </param>
80          /// <param name="header">
81          /// <para>A header.</para>
82          /// <para></para>
83          /// </param>
84          [MethodImpl(MethodImplOptions.AggressiveInlining)]
85          protected InternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLinkAddress>
            ↪ constants, byte* linksDataParts, byte* linksIndexParts, byte* header)
86          {
87              LinksDataParts = linksDataParts;
88              LinksIndexParts = linksIndexParts;
89              Header = header;
90              Break = constants.Break;
91              Continue = constants.Continue;
92          }
93
94          /// <summary>
95          /// <para>
96          /// Gets the tree root using the specified link.
97          /// </para>
98          /// <para></para>
99          /// </summary>
100         /// <param name="link">
101         /// <para>The link.</para>
102         /// <para></para>
103         /// </param>
104         /// <returns>
105         /// <para>The link</para>
106         /// <para></para>
107         /// </returns>
108         [MethodImpl(MethodImplOptions.AggressiveInlining)]
109         protected abstract TLinkAddress GetTreeRoot(TLinkAddress link);
110
111         /// <summary>
```

```
        /// <para>
        /// Gets the base part value using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TLinkAddress GetBasePartValue(TLinkAddress link);

        /// <summary>
        /// <para>
        /// Gets the key part value using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TLinkAddress GetKeyPartValue(TLinkAddress link);

        /// <summary>
        /// <para>
        /// Gets the link data part reference using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link data part of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref RawLinkDataPart<TLinkAddress>
        ↪  GetLinkDataPartReference(TLinkAddress link) => ref
        ↪  AsRef<RawLinkDataPart<TLinkAddress>>(LinksDataParts +
        ↪  (RawLinkDataPart<TLinkAddress>.SizeInBytes *
        ↪  _addressToInt64Converter.Convert(link)));

        /// <summary>
        /// <para>
        /// Gets the link index part reference using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link index part of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref RawLinkIndexPart<TLinkAddress>
        ↪  GetLinkIndexPartReference(TLinkAddress link) => ref
        ↪  AsRef<RawLinkIndexPart<TLinkAddress>>(LinksIndexParts +
        ↪  (RawLinkIndexPart<TLinkAddress>.SizeInBytes *
        ↪  _addressToInt64Converter.Convert(link)));

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLinkAddress first, TLinkAddress
        ↪  second) => LessThan(GetKeyPartValue(first), GetKeyPartValue(second));

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the right of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLinkAddress first, TLinkAddress
        ↪  second) => GreaterThan(GetKeyPartValue(first), GetKeyPartValue(second));

        /// <summary>
        /// <para>
        /// Gets the link values using the specified link index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="linkIndex">
        /// <para>The link index.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A list of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual IList<TLinkAddress>? GetLinkValues(TLinkAddress linkIndex)
        {
            ref var link = ref GetLinkDataPartReference(linkIndex);
            return new Link<TLinkAddress>(linkIndex, link.Source, link.Target);
        }

        /// <summary>
        /// <para>
        /// The zero.
        /// </para>
        /// <para></para>
        /// </summary>
        public TLinkAddress this[TLinkAddress link, TLinkAddress index]
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get
            {
                var root = GetTreeRoot(link);
                if (GreaterOrEqualThan(index, GetSize(root)))
                {
                    return Zero;
                }
```

```
258                 while (!EqualToZero(root))
259                 {
260                     var left = GetLeftOrDefault(root);
261                     var leftSize = GetSizeOrZero(left);
262                     if (LessThan(index, leftSize))
263                     {
264                         root = left;
265                         continue;
266                     }
267                     if (AreEqual(index, leftSize))
268                     {
269                         return root;
270                     }
271                     root = GetRightOrDefault(root);
272                     index = Subtract(index, Increment(leftSize));
273                 }
274                 return Zero; // TODO: Impossible situation exception (only if tree structure
                    ↪ broken)
275             }
276         }
277 
278         /// <summary>
279         /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
                ↪ (концом).
280         /// </summary>
281         /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
282         /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
283         /// <returns>Индекс искомой связи.</returns>
284         [MethodImpl(MethodImplOptions.AggressiveInlining)]
285         public abstract TLinkAddress Search(TLinkAddress source, TLinkAddress target);
286 
287         /// <summary>
288         /// <para>
289         /// Searches the core using the specified root.
290         /// </para>
291         /// <para></para>
292         /// </summary>
293         /// <param name="root">
294         /// <para>The root.</para>
295         /// <para></para>
296         /// </param>
297         /// <param name="key">
298         /// <para>The key.</para>
299         /// <para></para>
300         /// </param>
301         /// <returns>
302         /// <para>The zero.</para>
303         /// <para></para>
304         /// </returns>
305         [MethodImpl(MethodImplOptions.AggressiveInlining)]
306         protected TLinkAddress SearchCore(TLinkAddress root, TLinkAddress key)
307         {
308             while (!EqualToZero(root))
309             {
310                 var rootKey = GetKeyPartValue(root);
311                 if (LessThan(key, rootKey)) // node.Key < root.Key
312                 {
313                     root = GetLeftOrDefault(root);
314                 }
315                 else if (GreaterThan(key, rootKey)) // node.Key > root.Key
316                 {
317                     root = GetRightOrDefault(root);
318                 }
319                 else // node.Key == root.Key
320                 {
321                     return root;
322                 }
323             }
324             return Zero;
325         }
326 
327         // TODO: Return indices range instead of references count
328         /// <summary>
329         /// <para>
330         /// Counts the usages using the specified link.
331         /// </para>
332         /// <para></para>
333         /// </summary>
```

```
334         /// <param name="link">
335         /// <para>The link.</para>
336         /// <para></para>
337         /// </param>
338         /// <returns>
339         /// <para>The link</para>
340         /// <para></para>
341         /// </returns>
342         [MethodImpl(MethodImplOptions.AggressiveInlining)]
343         public TLinkAddress CountUsages(TLinkAddress link) => GetSizeOrZero(GetTreeRoot(link));
344 
345         /// <summary>
346         /// <para>
347         /// Eaches the usage using the specified base.
348         /// </para>
349         /// <para></para>
350         /// </summary>
351         /// <param name="@base">
352         /// <para>The base.</para>
353         /// <para></para>
354         /// </param>
355         /// <param name="handler">
356         /// <para>The handler.</para>
357         /// <para></para>
358         /// </param>
359         /// <returns>
360         /// <para>The link</para>
361         /// <para></para>
362         /// </returns>
363         [MethodImpl(MethodImplOptions.AggressiveInlining)]
364         public TLinkAddress EachUsage(TLinkAddress @base, ReadHandler<TLinkAddress>? handler) =>
          ↪  EachUsageCore(@base, GetTreeRoot(@base), handler);
365 
366         // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
          ↪  low-level MSIL stack.
367         [MethodImpl(MethodImplOptions.AggressiveInlining)]
368         private TLinkAddress EachUsageCore(TLinkAddress @base, TLinkAddress link,
          ↪  ReadHandler<TLinkAddress>? handler)
369         {
370             var @continue = Continue;
371             if (EqualToZero(link))
372             {
373                 return @continue;
374             }
375             var @break = Break;
376             if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
377             {
378                 return @break;
379             }
380             if (AreEqual(handler(GetLinkValues(link)), @break))
381             {
382                 return @break;
383             }
384             if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
385             {
386                 return @break;
387             }
388             return @continue;
389         }
390 
391         /// <summary>
392         /// <para>
393         /// Prints the node value using the specified node.
394         /// </para>
395         /// <para></para>
396         /// </summary>
397         /// <param name="node">
398         /// <para>The node.</para>
399         /// <para></para>
400         /// </param>
401         /// <param name="sb">
402         /// <para>The sb.</para>
403         /// <para></para>
404         /// </param>
405         [MethodImpl(MethodImplOptions.AggressiveInlining)]
406         protected override void PrintNodeValue(TLinkAddress node, StringBuilder sb)
407         {
408             ref var link = ref GetLinkDataPartReference(node);
```

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

1.40 ./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 Platform.Delegates;
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
{
    /// <summary>
    /// <para>
    /// Represents the internal links sources linked list methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="RelativeCircularDoublyLinkedListMethods{TLinkAddress}"/>
    public unsafe class InternalLinksSourcesLinkedListMethods<TLinkAddress> : RelativeCircularDoublyLinkedListMethods<TLinkAddress>
    {
        private static readonly UncheckedConverter<TLinkAddress, long> _addressToInt64Converter = UncheckedConverter<TLinkAddress, long>.Default;
        private readonly byte* _linksDataParts;
        private readonly byte* _linksIndexParts;
        /// <summary>
        /// <para>
        /// The break.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly TLinkAddress Break;
        /// <summary>
        /// <para>
        /// The continue.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly TLinkAddress Continue;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="InternalLinksSourcesLinkedListMethods"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksDataParts">
        /// <para>A links data parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksIndexParts">
        /// <para>A links index parts.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public InternalLinksSourcesLinkedListMethods(LinksConstants<TLinkAddress> constants, byte* linksDataParts, byte* linksIndexParts)
        {
            _linksDataParts = linksDataParts;
            _linksIndexParts = linksIndexParts;
            Break = constants.Break;
            Continue = constants.Continue;
        }

```

```
        /// <summary>
        /// <para>
        /// Gets the link data part reference using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link data part of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref RawLinkDataPart<TLinkAddress>
        ↪  GetLinkDataPartReference(TLinkAddress link) => ref
        ↪  AsRef<RawLinkDataPart<TLinkAddress>>(_linksDataParts +
        ↪  (RawLinkDataPart<TLinkAddress>.SizeInBytes *
        ↪  _addressToInt64Converter.Convert(link)));

        /// <summary>
        /// <para>
        /// Gets the link index part reference using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link index part of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref RawLinkIndexPart<TLinkAddress>
        ↪  GetLinkIndexPartReference(TLinkAddress link) => ref
        ↪  AsRef<RawLinkIndexPart<TLinkAddress>>(_linksIndexParts +
        ↪  (RawLinkIndexPart<TLinkAddress>.SizeInBytes *
        ↪  _addressToInt64Converter.Convert(link)));

        /// <summary>
        /// <para>
        /// Gets the first using the specified head.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="head">
        /// <para>The head.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetFirst(TLinkAddress head) =>
        ↪  GetLinkIndexPartReference(head).RootAsSource;

        /// <summary>
        /// <para>
        /// Gets the last using the specified head.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="head">
        /// <para>The head.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetLast(TLinkAddress head)
        {
            var first = GetLinkIndexPartReference(head).RootAsSource;
```

```
            if (EqualToZero(first))
            {
                return first;
            }
            else
            {
                return GetPrevious(first);
            }
        }

        /// <summary>
        /// <para>
        /// Gets the previous using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetPrevious(TLinkAddress element) =>
        ↪  GetLinkIndexPartReference(element).LeftAsSource;

        /// <summary>
        /// <para>
        /// Gets the next using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetNext(TLinkAddress element) =>
        ↪  GetLinkIndexPartReference(element).RightAsSource;

        /// <summary>
        /// <para>
        /// Gets the size using the specified head.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="head">
        /// <para>The head.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetSize(TLinkAddress head) =>
        ↪  GetLinkIndexPartReference(head).SizeAsSource;

        /// <summary>
        /// <para>
        /// Sets the first using the specified head.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="head">
        /// <para>The head.</para>
        /// <para></para>
        /// </param>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
```

```
211             [MethodImpl(MethodImplOptions.AggressiveInlining)]
212             protected override void SetFirst(TLinkAddress head, TLinkAddress element) =>
                ↪  GetLinkIndexPartReference(head).RootAsSource = element;
213
214             /// <summary>
215             /// <para>
216             /// Sets the last using the specified head.
217             /// </para>
218             /// <para></para>
219             /// </summary>
220             /// <param name="head">
221             /// <para>The head.</para>
222             /// <para></para>
223             /// </param>
224             /// <param name="element">
225             /// <para>The element.</para>
226             /// <para></para>
227             /// </param>
228             [MethodImpl(MethodImplOptions.AggressiveInlining)]
229             protected override void SetLast(TLinkAddress head, TLinkAddress element)
230             {
231                 //var first = GetLinkIndexPartReference(head).RootAsSource;
232                 //if (EqualToZero(first))
233                 //{
234                 //    SetFirst(head, element);
235                 //}
236                 //else
237                 //{
238                 //    SetPrevious(first, element);
239                 //}
240             }
241
242             /// <summary>
243             /// <para>
244             /// Sets the previous using the specified element.
245             /// </para>
246             /// <para></para>
247             /// </summary>
248             /// <param name="element">
249             /// <para>The element.</para>
250             /// <para></para>
251             /// </param>
252             /// <param name="previous">
253             /// <para>The previous.</para>
254             /// <para></para>
255             /// </param>
256             [MethodImpl(MethodImplOptions.AggressiveInlining)]
257             protected override void SetPrevious(TLinkAddress element, TLinkAddress previous) =>
                ↪  GetLinkIndexPartReference(element).LeftAsSource = previous;
258
259             /// <summary>
260             /// <para>
261             /// Sets the next using the specified element.
262             /// </para>
263             /// <para></para>
264             /// </summary>
265             /// <param name="element">
266             /// <para>The element.</para>
267             /// <para></para>
268             /// </param>
269             /// <param name="next">
270             /// <para>The next.</para>
271             /// <para></para>
272             /// </param>
273             [MethodImpl(MethodImplOptions.AggressiveInlining)]
274             protected override void SetNext(TLinkAddress element, TLinkAddress next) =>
                ↪  GetLinkIndexPartReference(element).RightAsSource = next;
275
276             /// <summary>
277             /// <para>
278             /// Sets the size using the specified head.
279             /// </para>
280             /// <para></para>
281             /// </summary>
282             /// <param name="head">
283             /// <para>The head.</para>
284             /// <para></para>
285             /// </param>
```

```
        /// <param name="size">
        /// <para>The size.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLinkAddress head, TLinkAddress size) =>
        ↪  GetLinkIndexPartReference(head).SizeAsSource = size;

        /// <summary>
        /// <para>
        /// Counts the usages using the specified head.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="head">
        /// <para>The head.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress CountUsages(TLinkAddress head) => GetSize(head);

        /// <summary>
        /// <para>
        /// Gets the link values using the specified link index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="linkIndex">
        /// <para>The link index.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A list of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual IList<TLinkAddress>? GetLinkValues(TLinkAddress linkIndex)
        {
            ref var link = ref GetLinkDataPartReference(linkIndex);
            return new Link<TLinkAddress>(linkIndex, link.Source, link.Target);
        }

        /// <summary>
        /// <para>
        /// Eaches the usage using the specified source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="handler">
        /// <para>The handler.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The continue.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress EachUsage(TLinkAddress source, ReadHandler<TLinkAddress>? 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;
        }
    }
}
```

1.41 ./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
{
    /// <summary>
    /// <para>
    /// Represents the internal links sources recursionless size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="InternalLinksRecursionlessSizeBalancedTreeMethodsBase{TLinkAddress}"/>
    public unsafe class InternalLinksSourcesRecursionlessSizeBalancedTreeMethods<TLinkAddress> :
    ↪  InternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see
        ↪  cref="InternalLinksSourcesRecursionlessSizeBalancedTreeMethods"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksDataParts">
        /// <para>A links data parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksIndexParts">
        /// <para>A links index parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<TLinkAddr⌋
        ↪  ess> constants, byte* linksDataParts, byte* linksIndexParts, byte* header) :
        ↪  base(constants, linksDataParts, linksIndexParts, header) { }

        /// <summary>
        /// <para>
        /// Gets the left reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
        ↪  GetLinkIndexPartReference(node).LeftAsSource;

        /// <summary>
        /// <para>
        /// Gets the right reference using the specified node.
        /// </para>
        /// <para></para>
```

```
63          /// </summary>
64          /// <param name="node">
65          /// <para>The node.</para>
66          /// <para></para>
67          /// </param>
68          /// <returns>
69          /// <para>The ref link</para>
70          /// <para></para>
71          /// </returns>
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
            ↪ GetLinkIndexPartReference(node).RightAsSource;
74
75          /// <summary>
76          /// <para>
77          /// Gets the left using the specified node.
78          /// </para>
79          /// <para></para>
80          /// </summary>
81          /// <param name="node">
82          /// <para>The node.</para>
83          /// <para></para>
84          /// </param>
85          /// <returns>
86          /// <para>The link</para>
87          /// <para></para>
88          /// </returns>
89          [MethodImpl(MethodImplOptions.AggressiveInlining)]
90          protected override TLinkAddress GetLeft(TLinkAddress node) =>
            ↪ GetLinkIndexPartReference(node).LeftAsSource;
91
92          /// <summary>
93          /// <para>
94          /// Gets the right using the specified node.
95          /// </para>
96          /// <para></para>
97          /// </summary>
98          /// <param name="node">
99          /// <para>The node.</para>
100         /// <para></para>
101         /// </param>
102         /// <returns>
103         /// <para>The link</para>
104         /// <para></para>
105         /// </returns>
106         [MethodImpl(MethodImplOptions.AggressiveInlining)]
107         protected override TLinkAddress GetRight(TLinkAddress node) =>
            ↪ GetLinkIndexPartReference(node).RightAsSource;
108
109         /// <summary>
110         /// <para>
111         /// Sets the left using the specified node.
112         /// </para>
113         /// <para></para>
114         /// </summary>
115         /// <param name="node">
116         /// <para>The node.</para>
117         /// <para></para>
118         /// </param>
119         /// <param name="left">
120         /// <para>The left.</para>
121         /// <para></para>
122         /// </param>
123         [MethodImpl(MethodImplOptions.AggressiveInlining)]
124         protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
            ↪ GetLinkIndexPartReference(node).LeftAsSource = left;
125
126         /// <summary>
127         /// <para>
128         /// Sets the right using the specified node.
129         /// </para>
130         /// <para></para>
131         /// </summary>
132         /// <param name="node">
133         /// <para>The node.</para>
134         /// <para></para>
135         /// </param>
136         /// <param name="right">
```

```
137             /// <para>The right.</para>
138             /// <para></para>
139             /// </param>
140             [MethodImpl(MethodImplOptions.AggressiveInlining)]
141             protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
                ↪ GetLinkIndexPartReference(node).RightAsSource = right;
142
143             /// <summary>
144             /// <para>
145             /// Gets the size using the specified node.
146             /// </para>
147             /// <para></para>
148             /// </summary>
149             /// <param name="node">
150             /// <para>The node.</para>
151             /// <para></para>
152             /// </param>
153             /// <returns>
154             /// <para>The link</para>
155             /// <para></para>
156             /// </returns>
157             [MethodImpl(MethodImplOptions.AggressiveInlining)]
158             protected override TLinkAddress GetSize(TLinkAddress node) =>
                ↪ GetLinkIndexPartReference(node).SizeAsSource;
159
160             /// <summary>
161             /// <para>
162             /// Sets the size using the specified node.
163             /// </para>
164             /// <para></para>
165             /// </summary>
166             /// <param name="node">
167             /// <para>The node.</para>
168             /// <para></para>
169             /// </param>
170             /// <param name="size">
171             /// <para>The size.</para>
172             /// <para></para>
173             /// </param>
174             [MethodImpl(MethodImplOptions.AggressiveInlining)]
175             protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
                ↪ GetLinkIndexPartReference(node).SizeAsSource = size;
176
177             /// <summary>
178             /// <para>
179             /// Gets the tree root using the specified link.
180             /// </para>
181             /// <para></para>
182             /// </summary>
183             /// <param name="link">
184             /// <para>The link.</para>
185             /// <para></para>
186             /// </param>
187             /// <returns>
188             /// <para>The link</para>
189             /// <para></para>
190             /// </returns>
191             [MethodImpl(MethodImplOptions.AggressiveInlining)]
192             protected override TLinkAddress GetTreeRoot(TLinkAddress link) =>
                ↪ GetLinkIndexPartReference(link).RootAsSource;
193
194             /// <summary>
195             /// <para>
196             /// Gets the base part value using the specified link.
197             /// </para>
198             /// <para></para>
199             /// </summary>
200             /// <param name="link">
201             /// <para>The link.</para>
202             /// <para></para>
203             /// </param>
204             /// <returns>
205             /// <para>The link</para>
206             /// <para></para>
207             /// </returns>
208             [MethodImpl(MethodImplOptions.AggressiveInlining)]
209             protected override TLinkAddress GetBasePartValue(TLinkAddress link) =>
                ↪ GetLinkDataPartReference(link).Source;
```

```

        /// <summary>
        /// <para>
        /// Gets the key part value using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetKeyPartValue(TLinkAddress link) =>
        ↪ GetLinkDataPartReference(link).Target;

        /// <summary>
        /// <para>
        /// Clears the node using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLinkAddress node)
        {
            ref var link = ref GetLinkIndexPartReference(node);
            link.LeftAsSource = Zero;
            link.RightAsSource = Zero;
            link.SizeAsSource = Zero;
        }

        /// <summary>
        /// <para>
        /// Searches the source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public override TLinkAddress Search(TLinkAddress source, TLinkAddress target) =>
        ↪ SearchCore(GetTreeRoot(source), target);
    }
}
```

1.42 ./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
{
    /// <summary>
    /// <para>
    /// Represents the internal links sources size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="InternalLinksSizeBalancedTreeMethodsBase{TLinkAddress}"/>
    public unsafe class InternalLinksSourcesSizeBalancedTreeMethods<TLinkAddress> :
    ↪ InternalLinksSizeBalancedTreeMethodsBase<TLinkAddress>
    {
```

```
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="InternalLinksSourcesSizeBalancedTreeMethods"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksDataParts">
        /// <para>A links data parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksIndexParts">
        /// <para>A links index parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public InternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLinkAddress>
        ↪  constants, byte* linksDataParts, byte* linksIndexParts, byte* header) :
        ↪  base(constants, linksDataParts, linksIndexParts, header) { }

        /// <summary>
        /// <para>
        /// Gets the left reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
        ↪  GetLinkIndexPartReference(node).LeftAsSource;

        /// <summary>
        /// <para>
        /// Gets the right reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
        ↪  GetLinkIndexPartReference(node).RightAsSource;

        /// <summary>
        /// <para>
        /// Gets the left using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
```

```
89          [MethodImpl(MethodImplOptions.AggressiveInlining)]
90          protected override TLinkAddress GetLeft(TLinkAddress node) =>
            ↪ GetLinkIndexPartReference(node).LeftAsSource;
91
92          /// <summary>
93          /// <para>
94          /// Gets the right using the specified node.
95          /// </para>
96          /// <para></para>
97          /// </summary>
98          /// <param name="node">
99          /// <para>The node.</para>
100         /// <para></para>
101         /// </param>
102         /// <returns>
103         /// <para>The link</para>
104         /// <para></para>
105         /// </returns>
106         [MethodImpl(MethodImplOptions.AggressiveInlining)]
107         protected override TLinkAddress GetRight(TLinkAddress node) =>
            ↪ GetLinkIndexPartReference(node).RightAsSource;
108
109         /// <summary>
110         /// <para>
111         /// Sets the left using the specified node.
112         /// </para>
113         /// <para></para>
114         /// </summary>
115         /// <param name="node">
116         /// <para>The node.</para>
117         /// <para></para>
118         /// </param>
119         /// <param name="left">
120         /// <para>The left.</para>
121         /// <para></para>
122         /// </param>
123         [MethodImpl(MethodImplOptions.AggressiveInlining)]
124         protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
            ↪ GetLinkIndexPartReference(node).LeftAsSource = left;
125
126         /// <summary>
127         /// <para>
128         /// Sets the right using the specified node.
129         /// </para>
130         /// <para></para>
131         /// </summary>
132         /// <param name="node">
133         /// <para>The node.</para>
134         /// <para></para>
135         /// </param>
136         /// <param name="right">
137         /// <para>The right.</para>
138         /// <para></para>
139         /// </param>
140         [MethodImpl(MethodImplOptions.AggressiveInlining)]
141         protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
            ↪ GetLinkIndexPartReference(node).RightAsSource = right;
142
143         /// <summary>
144         /// <para>
145         /// Gets the size using the specified node.
146         /// </para>
147         /// <para></para>
148         /// </summary>
149         /// <param name="node">
150         /// <para>The node.</para>
151         /// <para></para>
152         /// </param>
153         /// <returns>
154         /// <para>The link</para>
155         /// <para></para>
156         /// </returns>
157         [MethodImpl(MethodImplOptions.AggressiveInlining)]
158         protected override TLinkAddress GetSize(TLinkAddress node) =>
            ↪ GetLinkIndexPartReference(node).SizeAsSource;
159
160         /// <summary>
```

```
        /// <para>
        /// Sets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="size">
        /// <para>The size.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
        ↪ GetLinkIndexPartReference(node).SizeAsSource = size;

        /// <summary>
        /// <para>
        /// Gets the tree root using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetTreeRoot(TLinkAddress link) =>
        ↪ GetLinkIndexPartReference(link).RootAsSource;

        /// <summary>
        /// <para>
        /// Gets the base part value using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetBasePartValue(TLinkAddress link) =>
        ↪ GetLinkDataPartReference(link).Source;

        /// <summary>
        /// <para>
        /// Gets the key part value using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetKeyPartValue(TLinkAddress link) =>
        ↪ GetLinkDataPartReference(link).Target;

        /// <summary>
        /// <para>
        /// Clears the node using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
```

```
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLinkAddress node)
        {
            ref var link = ref GetLinkIndexPartReference(node);
            link.LeftAsSource = Zero;
            link.RightAsSource = Zero;
            link.SizeAsSource = Zero;
        }

        /// <summary>
        /// <para>
        /// Searches the source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public override TLinkAddress Search(TLinkAddress source, TLinkAddress target) =>
        ↪ SearchCore(GetTreeRoot(source), target);
    }
}
```

1.43 ./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
{
    /// <summary>
    /// <para>
    /// Represents the internal links targets recursionless size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="InternalLinksRecursionlessSizeBalancedTreeMethodsBase{TLinkAddress}"/>
    public unsafe class InternalLinksTargetsRecursionlessSizeBalancedTreeMethods<TLinkAddress> :
    ↪ InternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see
        ↪ cref="InternalLinksTargetsRecursionlessSizeBalancedTreeMethods"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksDataParts">
        /// <para>A links data parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksIndexParts">
        /// <para>A links index parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
39          public InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<TLinkAddr
            ↪ ess> constants, byte* linksDataParts, byte* linksIndexParts, byte* header) :
            ↪ base(constants, linksDataParts, linksIndexParts, header) { }
40
41          /// <summary>
42          /// <para>
43          /// Gets the left reference using the specified node.
44          /// </para>
45          /// <para></para>
46          /// </summary>
47          /// <param name="node">
48          /// <para>The node.</para>
49          /// <para></para>
50          /// </param>
51          /// <returns>
52          /// <para>The ref link</para>
53          /// <para></para>
54          /// </returns>
55          [MethodImpl(MethodImplOptions.AggressiveInlining)]
56          protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
            ↪ GetLinkIndexPartReference(node).LeftAsTarget;
57
58          /// <summary>
59          /// <para>
60          /// Gets the right reference using the specified node.
61          /// </para>
62          /// <para></para>
63          /// </summary>
64          /// <param name="node">
65          /// <para>The node.</para>
66          /// <para></para>
67          /// </param>
68          /// <returns>
69          /// <para>The ref link</para>
70          /// <para></para>
71          /// </returns>
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
            ↪ GetLinkIndexPartReference(node).RightAsTarget;
74
75          /// <summary>
76          /// <para>
77          /// Gets the left using the specified node.
78          /// </para>
79          /// <para></para>
80          /// </summary>
81          /// <param name="node">
82          /// <para>The node.</para>
83          /// <para></para>
84          /// </param>
85          /// <returns>
86          /// <para>The link</para>
87          /// <para></para>
88          /// </returns>
89          [MethodImpl(MethodImplOptions.AggressiveInlining)]
90          protected override TLinkAddress GetLeft(TLinkAddress node) =>
            ↪ GetLinkIndexPartReference(node).LeftAsTarget;
91
92          /// <summary>
93          /// <para>
94          /// Gets the right using the specified node.
95          /// </para>
96          /// <para></para>
97          /// </summary>
98          /// <param name="node">
99          /// <para>The node.</para>
100         /// <para></para>
101         /// </param>
102         /// <returns>
103         /// <para>The link</para>
104         /// <para></para>
105         /// </returns>
106         [MethodImpl(MethodImplOptions.AggressiveInlining)]
107         protected override TLinkAddress GetRight(TLinkAddress node) =>
            ↪ GetLinkIndexPartReference(node).RightAsTarget;
108
109         /// <summary>
```

```
110         /// <para>
111         /// Sets the left using the specified node.
112         /// </para>
113         /// <para></para>
114         /// </summary>
115         /// <param name="node">
116         /// <para>The node.</para>
117         /// <para></para>
118         /// </param>
119         /// <param name="left">
120         /// <para>The left.</para>
121         /// <para></para>
122         /// </param>
123         [MethodImpl(MethodImplOptions.AggressiveInlining)]
124         protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
            ↪ GetLinkIndexPartReference(node).LeftAsTarget = left;
125
126         /// <summary>
127         /// <para>
128         /// Sets the right using the specified node.
129         /// </para>
130         /// <para></para>
131         /// </summary>
132         /// <param name="node">
133         /// <para>The node.</para>
134         /// <para></para>
135         /// </param>
136         /// <param name="right">
137         /// <para>The right.</para>
138         /// <para></para>
139         /// </param>
140         [MethodImpl(MethodImplOptions.AggressiveInlining)]
141         protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
            ↪ GetLinkIndexPartReference(node).RightAsTarget = right;
142
143         /// <summary>
144         /// <para>
145         /// Gets the size using the specified node.
146         /// </para>
147         /// <para></para>
148         /// </summary>
149         /// <param name="node">
150         /// <para>The node.</para>
151         /// <para></para>
152         /// </param>
153         /// <returns>
154         /// <para>The link</para>
155         /// <para></para>
156         /// </returns>
157         [MethodImpl(MethodImplOptions.AggressiveInlining)]
158         protected override TLinkAddress GetSize(TLinkAddress node) =>
            ↪ GetLinkIndexPartReference(node).SizeAsTarget;
159
160         /// <summary>
161         /// <para>
162         /// Sets the size using the specified node.
163         /// </para>
164         /// <para></para>
165         /// </summary>
166         /// <param name="node">
167         /// <para>The node.</para>
168         /// <para></para>
169         /// </param>
170         /// <param name="size">
171         /// <para>The size.</para>
172         /// <para></para>
173         /// </param>
174         [MethodImpl(MethodImplOptions.AggressiveInlining)]
175         protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
            ↪ GetLinkIndexPartReference(node).SizeAsTarget = size;
176
177         /// <summary>
178         /// <para>
179         /// Gets the tree root using the specified link.
180         /// </para>
181         /// <para></para>
182         /// </summary>
183         /// <param name="link">
```

```
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetTreeRoot(TLinkAddress link) =>
        ↪ GetLinkIndexPartReference(link).RootAsTarget;

        /// <summary>
        /// <para>
        /// Gets the base part value using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetBasePartValue(TLinkAddress link) =>
        ↪ GetLinkDataPartReference(link).Target;

        /// <summary>
        /// <para>
        /// Gets the key part value using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetKeyPartValue(TLinkAddress link) =>
        ↪ GetLinkDataPartReference(link).Source;

        /// <summary>
        /// <para>
        /// Clears the node using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLinkAddress node)
        {
            ref var link = ref GetLinkIndexPartReference(node);
            link.LeftAsTarget = Zero;
            link.RightAsTarget = Zero;
            link.SizeAsTarget = Zero;
        }

        /// <summary>
        /// <para>
        /// Searches the source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
```

```
259         /// <para></para>
260         /// </param>
261         /// <returns>
262         /// <para>The link</para>
263         /// <para></para>
264         /// </returns>
265         public override TLinkAddress Search(TLinkAddress source, TLinkAddress target) =>
            ↪ SearchCore(GetTreeRoot(target), source);
266     }
267 }
```

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

```
1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Memory.Split.Generic
6  {
7      /// <summary>
8      /// <para>
9      /// Represents the internal links targets size balanced tree methods.
10     /// </para>
11     /// <para></para>
12     /// </summary>
13     /// <seealso cref="InternalLinksSizeBalancedTreeMethodsBase{TLinkAddress}"/>
14     public unsafe class InternalLinksTargetsSizeBalancedTreeMethods<TLinkAddress> :
       ↪ InternalLinksSizeBalancedTreeMethodsBase<TLinkAddress>
15     {
16         /// <summary>
17         /// <para>
18         /// Initializes a new <see cref="InternalLinksTargetsSizeBalancedTreeMethods"/> instance.
19         /// </para>
20         /// <para></para>
21         /// </summary>
22         /// <param name="constants">
23         /// <para>A constants.</para>
24         /// <para></para>
25         /// </param>
26         /// <param name="linksDataParts">
27         /// <para>A links data parts.</para>
28         /// <para></para>
29         /// </param>
30         /// <param name="linksIndexParts">
31         /// <para>A links index parts.</para>
32         /// <para></para>
33         /// </param>
34         /// <param name="header">
35         /// <para>A header.</para>
36         /// <para></para>
37         /// </param>
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public InternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLinkAddress>
           ↪ constants, byte* linksDataParts, byte* linksIndexParts, byte* header) :
           ↪ base(constants, linksDataParts, linksIndexParts, header) { }
40
41         /// <summary>
42         /// <para>
43         /// Gets the left reference using the specified node.
44         /// </para>
45         /// <para></para>
46         /// </summary>
47         /// <param name="node">
48         /// <para>The node.</para>
49         /// <para></para>
50         /// </param>
51         /// <returns>
52         /// <para>The ref link</para>
53         /// <para></para>
54         /// </returns>
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
           ↪ GetLinkIndexPartReference(node).LeftAsTarget;
57
58         /// <summary>
59         /// <para>
60         /// Gets the right reference using the specified node.
61         /// </para>
62         /// <para></para>
```

```
63          /// </summary>
64          /// <param name="node">
65          /// <para>The node.</para>
66          /// <para></para>
67          /// </param>
68          /// <returns>
69          /// <para>The ref link</para>
70          /// <para></para>
71          /// </returns>
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
            ↪ GetLinkIndexPartReference(node).RightAsTarget;
74
75          /// <summary>
76          /// <para>
77          /// Gets the left using the specified node.
78          /// </para>
79          /// <para></para>
80          /// </summary>
81          /// <param name="node">
82          /// <para>The node.</para>
83          /// <para></para>
84          /// </param>
85          /// <returns>
86          /// <para>The link</para>
87          /// <para></para>
88          /// </returns>
89          [MethodImpl(MethodImplOptions.AggressiveInlining)]
90          protected override TLinkAddress GetLeft(TLinkAddress node) =>
            ↪ GetLinkIndexPartReference(node).LeftAsTarget;
91
92          /// <summary>
93          /// <para>
94          /// Gets the right using the specified node.
95          /// </para>
96          /// <para></para>
97          /// </summary>
98          /// <param name="node">
99          /// <para>The node.</para>
100         /// <para></para>
101         /// </param>
102         /// <returns>
103         /// <para>The link</para>
104         /// <para></para>
105         /// </returns>
106         [MethodImpl(MethodImplOptions.AggressiveInlining)]
107         protected override TLinkAddress GetRight(TLinkAddress node) =>
            ↪ GetLinkIndexPartReference(node).RightAsTarget;
108
109         /// <summary>
110         /// <para>
111         /// Sets the left using the specified node.
112         /// </para>
113         /// <para></para>
114         /// </summary>
115         /// <param name="node">
116         /// <para>The node.</para>
117         /// <para></para>
118         /// </param>
119         /// <param name="left">
120         /// <para>The left.</para>
121         /// <para></para>
122         /// </param>
123         [MethodImpl(MethodImplOptions.AggressiveInlining)]
124         protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
            ↪ GetLinkIndexPartReference(node).LeftAsTarget = left;
125
126         /// <summary>
127         /// <para>
128         /// Sets the right using the specified node.
129         /// </para>
130         /// <para></para>
131         /// </summary>
132         /// <param name="node">
133         /// <para>The node.</para>
134         /// <para></para>
135         /// </param>
136         /// <param name="right">
```

```
137             /// <para>The right.</para>
138             /// <para></para>
139             /// </param>
140             [MethodImpl(MethodImplOptions.AggressiveInlining)]
141             protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
                ↪ GetLinkIndexPartReference(node).RightAsTarget = right;
142
143             /// <summary>
144             /// <para>
145             /// Gets the size using the specified node.
146             /// </para>
147             /// <para></para>
148             /// </summary>
149             /// <param name="node">
150             /// <para>The node.</para>
151             /// <para></para>
152             /// </param>
153             /// <returns>
154             /// <para>The link</para>
155             /// <para></para>
156             /// </returns>
157             [MethodImpl(MethodImplOptions.AggressiveInlining)]
158             protected override TLinkAddress GetSize(TLinkAddress node) =>
                ↪ GetLinkIndexPartReference(node).SizeAsTarget;
159
160             /// <summary>
161             /// <para>
162             /// Sets the size using the specified node.
163             /// </para>
164             /// <para></para>
165             /// </summary>
166             /// <param name="node">
167             /// <para>The node.</para>
168             /// <para></para>
169             /// </param>
170             /// <param name="size">
171             /// <para>The size.</para>
172             /// <para></para>
173             /// </param>
174             [MethodImpl(MethodImplOptions.AggressiveInlining)]
175             protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
                ↪ GetLinkIndexPartReference(node).SizeAsTarget = size;
176
177             /// <summary>
178             /// <para>
179             /// Gets the tree root using the specified link.
180             /// </para>
181             /// <para></para>
182             /// </summary>
183             /// <param name="link">
184             /// <para>The link.</para>
185             /// <para></para>
186             /// </param>
187             /// <returns>
188             /// <para>The link</para>
189             /// <para></para>
190             /// </returns>
191             [MethodImpl(MethodImplOptions.AggressiveInlining)]
192             protected override TLinkAddress GetTreeRoot(TLinkAddress link) =>
                ↪ GetLinkIndexPartReference(link).RootAsTarget;
193
194             /// <summary>
195             /// <para>
196             /// Gets the base part value using the specified link.
197             /// </para>
198             /// <para></para>
199             /// </summary>
200             /// <param name="link">
201             /// <para>The link.</para>
202             /// <para></para>
203             /// </param>
204             /// <returns>
205             /// <para>The link</para>
206             /// <para></para>
207             /// </returns>
208             [MethodImpl(MethodImplOptions.AggressiveInlining)]
209             protected override TLinkAddress GetBasePartValue(TLinkAddress link) =>
                ↪ GetLinkDataPartReference(link).Target;
```

```
210
211         /// <summary>
212         /// <para>
213         /// Gets the key part value using the specified link.
214         /// </para>
215         /// <para></para>
216         /// </summary>
217         /// <param name="link">
218         /// <para>The link.</para>
219         /// <para></para>
220         /// </param>
221         /// <returns>
222         /// <para>The link</para>
223         /// <para></para>
224         /// </returns>
225         [MethodImpl(MethodImplOptions.AggressiveInlining)]
226         protected override TLinkAddress GetKeyPartValue(TLinkAddress link) =>
            ↪ GetLinkDataPartReference(link).Source;
227
228         /// <summary>
229         /// <para>
230         /// Clears the node using the specified node.
231         /// </para>
232         /// <para></para>
233         /// </summary>
234         /// <param name="node">
235         /// <para>The node.</para>
236         /// <para></para>
237         /// </param>
238         [MethodImpl(MethodImplOptions.AggressiveInlining)]
239         protected override void ClearNode(TLinkAddress node)
240         {
241             ref var link = ref GetLinkIndexPartReference(node);
242             link.LeftAsTarget = Zero;
243             link.RightAsTarget = Zero;
244             link.SizeAsTarget = Zero;
245         }
246
247         /// <summary>
248         /// <para>
249         /// Searches the source.
250         /// </para>
251         /// <para></para>
252         /// </summary>
253         /// <param name="source">
254         /// <para>The source.</para>
255         /// <para></para>
256         /// </param>
257         /// <param name="target">
258         /// <para>The target.</para>
259         /// <para></para>
260         /// </param>
261         /// <returns>
262         /// <para>The link</para>
263         /// <para></para>
264         /// </returns>
265         public override TLinkAddress Search(TLinkAddress source, TLinkAddress target) =>
            ↪ SearchCore(GetTreeRoot(target), source);
266     }
267 }
```

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

```
1  using System;
2  using System.Runtime.CompilerServices;
3  using Platform.Singletons;
4  using Platform.Memory;
5  using static System.Runtime.CompilerServices.Unsafe;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Memory.Split.Generic
10 {
11     /// <summary>
12     /// <para>
13     /// Represents the split memory links.
14     /// </para>
15     /// <para></para>
16     /// </summary>
17     /// <seealso cref="SplitMemoryLinksBase{TLinkAddress}"/>
```

```
18      public unsafe class SplitMemoryLinks<TLinkAddress> : SplitMemoryLinksBase<TLinkAddress>
        ↪  where TLinkAddress : struct
19      {
20          private readonly Func<ILinksTreeMethods<TLinkAddress>> _createInternalSourceTreeMethods;
21          private readonly Func<ILinksTreeMethods<TLinkAddress>> _createExternalSourceTreeMethods;
22          private readonly Func<ILinksTreeMethods<TLinkAddress>> _createInternalTargetTreeMethods;
23          private readonly Func<ILinksTreeMethods<TLinkAddress>> _createExternalTargetTreeMethods;
24          private byte* _header;
25          private byte* _linksDataParts;
26          private byte* _linksIndexParts;
27
28          /// <summary>
29          /// <para>
30          /// Initializes a new <see cref="SplitMemoryLinks"/> instance.
31          /// </para>
32          /// <para></para>
33          /// </summary>
34          /// <param name="dataMemory">
35          /// <para>A data memory.</para>
36          /// <para></para>
37          /// </param>
38          /// <param name="indexMemory">
39          /// <para>A index memory.</para>
40          /// <para></para>
41          /// </param>
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          public SplitMemoryLinks(string dataMemory, string indexMemory) : this(new
            ↪  FileMappedResizableDirectMemory(dataMemory), new
            ↪  FileMappedResizableDirectMemory(indexMemory)) { }
44
45          /// <summary>
46          /// <para>
47          /// Initializes a new <see cref="SplitMemoryLinks"/> instance.
48          /// </para>
49          /// <para></para>
50          /// </summary>
51          /// <param name="dataMemory">
52          /// <para>A data memory.</para>
53          /// <para></para>
54          /// </param>
55          /// <param name="indexMemory">
56          /// <para>A index memory.</para>
57          /// <para></para>
58          /// </param>
59          [MethodImpl(MethodImplOptions.AggressiveInlining)]
60          public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪  indexMemory) : this(dataMemory, indexMemory, DefaultLinksSizeStep) { }
61
62          /// <summary>
63          /// <para>
64          /// Initializes a new <see cref="SplitMemoryLinks"/> instance.
65          /// </para>
66          /// <para></para>
67          /// </summary>
68          /// <param name="dataMemory">
69          /// <para>A data memory.</para>
70          /// <para></para>
71          /// </param>
72          /// <param name="indexMemory">
73          /// <para>A index memory.</para>
74          /// <para></para>
75          /// </param>
76          /// <param name="memoryReservationStep">
77          /// <para>A memory reservation step.</para>
78          /// <para></para>
79          /// </param>
80          [MethodImpl(MethodImplOptions.AggressiveInlining)]
81          public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪  indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
            ↪  memoryReservationStep, Default<LinksConstants<TLinkAddress>>.Instance,
            ↪  IndexTreeType.Default, useLinkedList: true) { }
82
83          /// <summary>
84          /// <para>
85          /// Initializes a new <see cref="SplitMemoryLinks"/> instance.
86          /// </para>
87          /// <para></para>
88          /// </summary>
```

```
89          /// <param name="dataMemory">
90          /// <para>A data memory.</para>
91          /// <para></para>
92          /// </param>
93          /// <param name="indexMemory">
94          /// <para>A index memory.</para>
95          /// <para></para>
96          /// </param>
97          /// <param name="memoryReservationStep">
98          /// <para>A memory reservation step.</para>
99          /// <para></para>
100         /// </param>
101         /// <param name="constants">
102         /// <para>A constants.</para>
103         /// <para></para>
104         /// </param>
105         [MethodImpl(MethodImplOptions.AggressiveInlining)]
106         public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪ indexMemory, long memoryReservationStep, LinksConstants<TLinkAddress> constants) :
            ↪ this(dataMemory, indexMemory, memoryReservationStep, constants,
            ↪ IndexTreeType.Default, useLinkedList: true) { }
107
108         /// <summary>
109         /// <para>
110         /// Initializes a new <see cref="SplitMemoryLinks"/> instance.
111         /// </para>
112         /// <para></para>
113         /// </summary>
114         /// <param name="dataMemory">
115         /// <para>A data memory.</para>
116         /// <para></para>
117         /// </param>
118         /// <param name="indexMemory">
119         /// <para>A index memory.</para>
120         /// <para></para>
121         /// </param>
122         /// <param name="memoryReservationStep">
123         /// <para>A memory reservation step.</para>
124         /// <para></para>
125         /// </param>
126         /// <param name="constants">
127         /// <para>A constants.</para>
128         /// <para></para>
129         /// </param>
130         /// <param name="indexTreeType">
131         /// <para>A index tree type.</para>
132         /// <para></para>
133         /// </param>
134         /// <param name="useLinkedList">
135         /// <para>A use linked list.</para>
136         /// <para></para>
137         /// </param>
138         [MethodImpl(MethodImplOptions.AggressiveInlining)]
139         public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪ indexMemory, long memoryReservationStep, LinksConstants<TLinkAddress> constants,
            ↪ IndexTreeType indexTreeType, bool useLinkedList) : base(dataMemory, indexMemory,
            ↪ memoryReservationStep, constants, useLinkedList)
140         {
141             if (indexTreeType == IndexTreeType.SizeBalancedTree)
142             {
143                 _createInternalSourceTreeMethods = () => new
                    ↪ InternalLinksSourcesSizeBalancedTreeMethods<TLinkAddress>(Constants,
                    ↪ _linksDataParts, _linksIndexParts, _header);
144                 _createExternalSourceTreeMethods = () => new
                    ↪ ExternalLinksSourcesSizeBalancedTreeMethods<TLinkAddress>(Constants,
                    ↪ _linksDataParts, _linksIndexParts, _header);
145                 _createInternalTargetTreeMethods = () => new
                    ↪ InternalLinksTargetsSizeBalancedTreeMethods<TLinkAddress>(Constants,
                    ↪ _linksDataParts, _linksIndexParts, _header);
146                 _createExternalTargetTreeMethods = () => new
                    ↪ ExternalLinksTargetsSizeBalancedTreeMethods<TLinkAddress>(Constants,
                    ↪ _linksDataParts, _linksIndexParts, _header);
147             }
148             else
149             {
```

```
150                 _createInternalSourceTreeMethods = () => new InternalLinksSourcesRecursionlessSi
                    ↪  zeBalancedTreeMethods<TLinkAddress>(Constants, _linksDataParts,
                    ↪  _linksIndexParts, _header);
151                 _createExternalSourceTreeMethods = () => new ExternalLinksSourcesRecursionlessSi
                    ↪  zeBalancedTreeMethods<TLinkAddress>(Constants, _linksDataParts,
                    ↪  _linksIndexParts, _header);
152                 _createInternalTargetTreeMethods = () => new InternalLinksTargetsRecursionlessSi
                    ↪  zeBalancedTreeMethods<TLinkAddress>(Constants, _linksDataParts,
                    ↪  _linksIndexParts, _header);
153                 _createExternalTargetTreeMethods = () => new ExternalLinksTargetsRecursionlessSi
                    ↪  zeBalancedTreeMethods<TLinkAddress>(Constants, _linksDataParts,
                    ↪  _linksIndexParts, _header);
154             }
155             Init(dataMemory, indexMemory);
156         }
157
158         /// <summary>
159         /// <para>
160         /// Sets the pointers using the specified data memory.
161         /// </para>
162         /// <para></para>
163         /// </summary>
164         /// <param name="dataMemory">
165         /// <para>The data memory.</para>
166         /// <para></para>
167         /// </param>
168         /// <param name="indexMemory">
169         /// <para>The index memory.</para>
170         /// <para></para>
171         /// </param>
172         [MethodImpl(MethodImplOptions.AggressiveInlining)]
173         protected override void SetPointers(IResizableDirectMemory dataMemory,
            ↪  IResizableDirectMemory indexMemory)
174         {
175             _linksDataParts = (byte*)dataMemory.Pointer;
176             _linksIndexParts = (byte*)indexMemory.Pointer;
177             _header = _linksIndexParts;
178             if (_useLinkedList)
179             {
180                 InternalSourcesListMethods = new
                    ↪  InternalLinksSourcesLinkedListMethods<TLinkAddress>(Constants,
                    ↪  _linksDataParts, _linksIndexParts);
181             }
182             else
183             {
184                 InternalSourcesTreeMethods = _createInternalSourceTreeMethods();
185             }
186             ExternalSourcesTreeMethods = _createExternalSourceTreeMethods();
187             InternalTargetsTreeMethods = _createInternalTargetTreeMethods();
188             ExternalTargetsTreeMethods = _createExternalTargetTreeMethods();
189             UnusedLinksListMethods = new UnusedLinksListMethods<TLinkAddress>(_linksDataParts,
                ↪  _header);
190         }
191
192         /// <summary>
193         /// <para>
194         /// Resets the pointers.
195         /// </para>
196         /// <para></para>
197         /// </summary>
198         [MethodImpl(MethodImplOptions.AggressiveInlining)]
199         protected override void ResetPointers()
200         {
201             base.ResetPointers();
202             _linksDataParts = null;
203             _linksIndexParts = null;
204             _header = null;
205         }
206
207         /// <summary>
208         /// <para>
209         /// Gets the header reference.
210         /// </para>
211         /// <para></para>
212         /// </summary>
213         /// <returns>
214         /// <para>A ref links header of t link</para>
215         /// <para></para>
```

```
216             /// </returns>
217             [MethodImpl(MethodImplOptions.AggressiveInlining)]
218             protected override ref LinksHeader<TLinkAddress> GetHeaderReference() => ref
                ↪  AsRef<LinksHeader<TLinkAddress>>(_header);
219
220             /// <summary>
221             /// <para>
222             /// Gets the link data part reference using the specified link index.
223             /// </para>
224             /// <para></para>
225             /// </summary>
226             /// <param name="linkIndex">
227             /// <para>The link index.</para>
228             /// <para></para>
229             /// </param>
230             /// <returns>
231             /// <para>A ref raw link data part of t link</para>
232             /// <para></para>
233             /// </returns>
234             [MethodImpl(MethodImplOptions.AggressiveInlining)]
235             protected override ref RawLinkDataPart<TLinkAddress>
                ↪  GetLinkDataPartReference(TLinkAddress linkIndex) => ref
                ↪  AsRef<RawLinkDataPart<TLinkAddress>>(_linksDataParts + (LinkDataPartSizeInBytes *
                ↪  ConvertToInt64(linkIndex)));
236
237             /// <summary>
238             /// <para>
239             /// Gets the link index part reference using the specified link index.
240             /// </para>
241             /// <para></para>
242             /// </summary>
243             /// <param name="linkIndex">
244             /// <para>The link index.</para>
245             /// <para></para>
246             /// </param>
247             /// <returns>
248             /// <para>A ref raw link index part of t link</para>
249             /// <para></para>
250             /// </returns>
251             [MethodImpl(MethodImplOptions.AggressiveInlining)]
252             protected override ref RawLinkIndexPart<TLinkAddress>
                ↪  GetLinkIndexPartReference(TLinkAddress linkIndex) => ref
                ↪  AsRef<RawLinkIndexPart<TLinkAddress>>(_linksIndexParts + (LinkIndexPartSizeInBytes *
                ↪  ConvertToInt64(linkIndex)));
253         }
254     }
```

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

```
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using Platform.Disposables;
5   using Platform.Singletons;
6   using Platform.Converters;
7   using Platform.Numbers;
8   using Platform.Memory;
9   using Platform.Data.Exceptions;
10  using Platform.Delegates;
11
12  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
13
14  namespace Platform.Data.Doublets.Memory.Split.Generic
15  {
16      /// <summary>
17      /// <para>
18      /// Represents the split memory links base.
19      /// </para>
20      /// <para></para>
21      /// </summary>
22      /// <seealso cref="DisposableBase"/>
23      /// <seealso cref="ILinks{TLinkAddress}"/>
24      public abstract class SplitMemoryLinksBase<TLinkAddress> : DisposableBase,
        ↪  ILinks<TLinkAddress> where TLinkAddress : struct
25      {
26          private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
            ↪  EqualityComparer<TLinkAddress>.Default;
27          private static readonly Comparer<TLinkAddress> _comparer =
            ↪  Comparer<TLinkAddress>.Default;
28          private static readonly UncheckedConverter<TLinkAddress, long> _addressToInt64Converter
            ↪  = UncheckedConverter<TLinkAddress, long>.Default;
```

```
29          private static readonly UncheckedConverter<long, TLinkAddress> _int64ToAddressConverter
            ↪  = UncheckedConverter<long, TLinkAddress>.Default;
30          private static readonly TLinkAddress _zero = default;
31          private static readonly TLinkAddress _one = Arithmetic.Increment(_zero);
32
33          /// <summary>Возвращает размер одной связи в байтах.</summary>
34          /// <remarks>
35          /// Используется только во вне класса, не рекомедуется использовать внутри.
36          /// Так как во вне не обязательно будет доступен unsafe C#.
37          /// </remarks>
38          public static readonly long LinkDataPartSizeInBytes =
            ↪  RawLinkDataPart<TLinkAddress>.SizeInBytes;
39
40          /// <summary>
41          /// <para>
42          /// The size in bytes.
43          /// </para>
44          /// <para></para>
45          /// </summary>
46          public static readonly long LinkIndexPartSizeInBytes =
            ↪  RawLinkIndexPart<TLinkAddress>.SizeInBytes;
47
48          /// <summary>
49          /// <para>
50          /// The size in bytes.
51          /// </para>
52          /// <para></para>
53          /// </summary>
54          public static readonly long LinkHeaderSizeInBytes =
            ↪  LinksHeader<TLinkAddress>.SizeInBytes;
55
56          /// <summary>
57          /// <para>
58          /// The default links size step.
59          /// </para>
60          /// <para></para>
61          /// </summary>
62          public static readonly long DefaultLinksSizeStep = 1 * 1024 * 1024;
63
64          /// <summary>
65          /// <para>
66          /// The data memory.
67          /// </para>
68          /// <para></para>
69          /// </summary>
70          protected readonly IResizableDirectMemory _dataMemory;
71          /// <summary>
72          /// <para>
73          /// The index memory.
74          /// </para>
75          /// <para></para>
76          /// </summary>
77          protected readonly IResizableDirectMemory _indexMemory;
78          /// <summary>
79          /// <para>
80          /// The use linked list.
81          /// </para>
82          /// <para></para>
83          /// </summary>
84          protected readonly bool _useLinkedList;
85          /// <summary>
86          /// <para>
87          /// The data memory reservation step in bytes.
88          /// </para>
89          /// <para></para>
90          /// </summary>
91          protected readonly long _dataMemoryReservationStepInBytes;
92          /// <summary>
93          /// <para>
94          /// The index memory reservation step in bytes.
95          /// </para>
96          /// <para></para>
97          /// </summary>
98          protected readonly long _indexMemoryReservationStepInBytes;
99
100         /// <summary>
101         /// <para>
102         /// The internal sources list methods.
103         /// </para>
```

```
104         /// <para></para>
105         /// </summary>
106         protected InternalLinksSourcesLinkedListMethods<TLinkAddress> InternalSourcesListMethods;
107         /// <summary>
108         /// <para>
109         /// The internal sources tree methods.
110         /// </para>
111         /// <para></para>
112         /// </summary>
113         protected ILinksTreeMethods<TLinkAddress> InternalSourcesTreeMethods;
114         /// <summary>
115         /// <para>
116         /// The external sources tree methods.
117         /// </para>
118         /// <para></para>
119         /// </summary>
120         protected ILinksTreeMethods<TLinkAddress> ExternalSourcesTreeMethods;
121         /// <summary>
122         /// <para>
123         /// The internal targets tree methods.
124         /// </para>
125         /// <para></para>
126         /// </summary>
127         protected ILinksTreeMethods<TLinkAddress> InternalTargetsTreeMethods;
128         /// <summary>
129         /// <para>
130         /// The external targets tree methods.
131         /// </para>
132         /// <para></para>
133         /// </summary>
134         protected ILinksTreeMethods<TLinkAddress> ExternalTargetsTreeMethods;
135         // 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
136         /// <summary>
137         /// <para>
138         /// The unused links list methods.
139         /// </para>
140         /// <para></para>
141         /// </summary>
142         protected ILinksListMethods<TLinkAddress> UnusedLinksListMethods;
143
144         /// <summary>
145         /// Возвращает общее число связей находящихся в хранилище.
146         /// </summary>
147         protected virtual TLinkAddress Total
148         {
149             [MethodImpl(MethodImplOptions.AggressiveInlining)]
150             get
151             {
152                 ref var header = ref GetHeaderReference();
153                 return Subtract(header.AllocatedLinks, header.FreeLinks);
154             }
155         }
156
157         /// <summary>
158         /// <para>
159         /// Gets the constants value.
160         /// </para>
161         /// <para></para>
162         /// </summary>
163         public virtual LinksConstants<TLinkAddress> Constants
164         {
165             [MethodImpl(MethodImplOptions.AggressiveInlining)]
166             get;
167         }
168
169         /// <summary>
170         /// <para>
171         /// Initializes a new <see cref="SplitMemoryLinksBase"/> instance.
172         /// </para>
173         /// <para></para>
174         /// </summary>
175         /// <param name="dataMemory">
176         /// <para>A data memory.</para>
177         /// <para></para>
178         /// </param>
179         /// <param name="indexMemory">
180         /// <para>A index memory.</para>
```

```
        /// <para></para>
        /// </param>
        /// <param name="memoryReservationStep">
        /// <para>A memory reservation step.</para>
        /// <para></para>
        /// </param>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="useLinkedList">
        /// <para>A use linked list.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
        ↪  indexMemory, long memoryReservationStep, LinksConstants<TLinkAddress> constants,
        ↪  bool useLinkedList)
        {
            _dataMemory = dataMemory;
            _indexMemory = indexMemory;
            _dataMemoryReservationStepInBytes = memoryReservationStep * LinkDataPartSizeInBytes;
            _indexMemoryReservationStepInBytes = memoryReservationStep *
            ↪  LinkIndexPartSizeInBytes;
            _useLinkedList = useLinkedList;
            Constants = constants;
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="SplitMemoryLinksBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="dataMemory">
        /// <para>A data memory.</para>
        /// <para></para>
        /// </param>
        /// <param name="indexMemory">
        /// <para>A index memory.</para>
        /// <para></para>
        /// </param>
        /// <param name="memoryReservationStep">
        /// <para>A memory reservation step.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
        ↪  indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
        ↪  memoryReservationStep, Default<LinksConstants<TLinkAddress>>.Instance,
        ↪  useLinkedList: true) { }

        /// <summary>
        /// <para>
        /// Inits the data memory.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="dataMemory">
        /// <para>The data memory.</para>
        /// <para></para>
        /// </param>
        /// <param name="indexMemory">
        /// <para>The index memory.</para>
        /// <para></para>
        /// </param>
        [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);
```

```
252             // Adjust reserved capacity
253             var minimumDataReservedCapacity = allocatedLinks * LinkDataPartSizeInBytes;
254             if (minimumDataReservedCapacity < dataMemory.UsedCapacity)
255             {
256                 minimumDataReservedCapacity = dataMemory.UsedCapacity;
257             }
258             if (minimumDataReservedCapacity < _dataMemoryReservationStepInBytes)
259             {
260                 minimumDataReservedCapacity = _dataMemoryReservationStepInBytes;
261             }
262             var minimumIndexReservedCapacity = allocatedLinks * LinkDataPartSizeInBytes;
263             if (minimumIndexReservedCapacity < indexMemory.UsedCapacity)
264             {
265                 minimumIndexReservedCapacity = indexMemory.UsedCapacity;
266             }
267             if (minimumIndexReservedCapacity < _indexMemoryReservationStepInBytes)
268             {
269                 minimumIndexReservedCapacity = _indexMemoryReservationStepInBytes;
270             }
271             // Check for alignment
272             if (minimumDataReservedCapacity % _dataMemoryReservationStepInBytes > 0)
273             {
274                 minimumDataReservedCapacity = ((minimumDataReservedCapacity /
                    ↪  _dataMemoryReservationStepInBytes) * _dataMemoryReservationStepInBytes) +
                    ↪  _dataMemoryReservationStepInBytes;
275             }
276             if (minimumIndexReservedCapacity % _indexMemoryReservationStepInBytes > 0)
277             {
278                 minimumIndexReservedCapacity = ((minimumIndexReservedCapacity /
                    ↪  _indexMemoryReservationStepInBytes) * _indexMemoryReservationStepInBytes) +
                    ↪  _indexMemoryReservationStepInBytes;
279             }
280             if (dataMemory.ReservedCapacity != minimumDataReservedCapacity)
281             {
282                 dataMemory.ReservedCapacity = minimumDataReservedCapacity;
283             }
284             if (indexMemory.ReservedCapacity != minimumIndexReservedCapacity)
285             {
286                 indexMemory.ReservedCapacity = minimumIndexReservedCapacity;
287             }
288             SetPointers(dataMemory, indexMemory);
289             header = ref GetHeaderReference();
290             // Ensure correctness _memory.UsedCapacity over _header->AllocatedLinks
291             // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
292             dataMemory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) *
                ↪  LinkDataPartSizeInBytes) + LinkDataPartSizeInBytes; // First link is read only
                ↪  zero link.
293             indexMemory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) *
                ↪  LinkIndexPartSizeInBytes) + LinkHeaderSizeInBytes;
294             // Ensure correctness _memory.ReservedLinks over _header->ReservedCapacity
295             // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
296             header.ReservedLinks = ConvertToAddress((dataMemory.ReservedCapacity -
                ↪  LinkDataPartSizeInBytes) / LinkDataPartSizeInBytes);
297         }
298
299         /// <summary>
300         /// <para>
301         /// Counts the substitution.
302         /// </para>
303         /// <para></para>
304         /// </summary>
305         /// <param name="restriction">
306         /// <para>The substitution.</para>
307         /// <para></para>
308         /// </param>
309         /// <exception cref="NotSupportedException">
310         /// <para>Другие размеры и способы ограничений не поддерживаются.</para>
311         /// <para></para>
312         /// </exception>
313         /// <returns>
314         /// <para>The link</para>
315         /// <para></para>
316         /// </returns>
317         [MethodImpl(MethodImplOptions.AggressiveInlining)]
318         public virtual TLinkAddress Count(IList<TLinkAddress>? restriction)
319         {
320             // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
321             if (restriction.Count == 0)
```

```
322                 {
323                     return Total;
324                 }
325                 var constants = Constants;
326                 var any = constants.Any;
327                 var index = restriction[constants.IndexPart];
328                 if (restriction.Count == 1)
329                 {
330                     if (AreEqual(index, any))
331                     {
332                         return Total;
333                     }
334                     return Exists(index) ? GetOne() : GetZero();
335                 }
336                 if (restriction.Count == 2)
337                 {
338                     var value = restriction[1];
339                     if (AreEqual(index, any))
340                     {
341                         if (AreEqual(value, any))
342                         {
343                             return Total; // Any - как отсутствие ограничения
344                         }
345                         var externalReferencesRange = constants.ExternalReferencesRange;
346                         if (externalReferencesRange.HasValue &&
                             ↪  externalReferencesRange.Value.Contains(value))
347                         {
348                             return Add(ExternalSourcesTreeMethods.CountUsages(value),
                                 ↪  ExternalTargetsTreeMethods.CountUsages(value));
349                         }
350                         else
351                         {
352                             if (_useLinkedList)
353                             {
354                                 return Add(InternalSourcesListMethods.CountUsages(value),
                                     ↪  InternalTargetsTreeMethods.CountUsages(value));
355                             }
356                             else
357                             {
358                                 return Add(InternalSourcesTreeMethods.CountUsages(value),
                                     ↪  InternalTargetsTreeMethods.CountUsages(value));
359                             }
360                         }
361                     }
362                     else
363                     {
364                         if (!Exists(index))
365                         {
366                             return GetZero();
367                         }
368                         if (AreEqual(value, any))
369                         {
370                             return GetOne();
371                         }
372                         ref var storedLinkValue = ref GetLinkDataPartReference(index);
373                         if (AreEqual(storedLinkValue.Source, value) ||
                             ↪  AreEqual(storedLinkValue.Target, value))
374                         {
375                             return GetOne();
376                         }
377                         return GetZero();
378                     }
379                 }
380                 if (restriction.Count == 3)
381                 {
382                     var externalReferencesRange = constants.ExternalReferencesRange;
383                     var source = restriction[constants.SourcePart];
384                     var target = restriction[constants.TargetPart];
385                     if (AreEqual(index, any))
386                     {
387                         if (AreEqual(source, any) && AreEqual(target, any))
388                         {
389                             return Total;
390                         }
391                         else if (AreEqual(source, any))
392                         {
393                             if (externalReferencesRange.HasValue &&
                                 ↪  externalReferencesRange.Value.Contains(target))
```

```
394                         {
395                             return ExternalTargetsTreeMethods.CountUsages(target);
396                         }
397                         else
398                         {
399                             return InternalTargetsTreeMethods.CountUsages(target);
400                         }
401                     }
402                     else if (AreEqual(target, any))
403                     {
404                         if (externalReferencesRange.HasValue &&
                            ↪  externalReferencesRange.Value.Contains(source))
405                         {
406                             return ExternalSourcesTreeMethods.CountUsages(source);
407                         }
408                         else
409                         {
410                             if (_useLinkedList)
411                             {
412                                 return InternalSourcesListMethods.CountUsages(source);
413                             }
414                             else
415                             {
416                                 return InternalSourcesTreeMethods.CountUsages(source);
417                             }
418                         }
419                     }
420                     else //if(source != Any && target != Any)
421                     {
422                         // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
423                         TLinkAddress link;
424                         if (externalReferencesRange.HasValue)
425                         {
426                             if (externalReferencesRange.Value.Contains(source) &&
                                ↪  externalReferencesRange.Value.Contains(target))
427                             {
428                                 link = ExternalSourcesTreeMethods.Search(source, target);
429                             }
430                             else if (externalReferencesRange.Value.Contains(source))
431                             {
432                                 link = InternalTargetsTreeMethods.Search(source, target);
433                             }
434                             else if (externalReferencesRange.Value.Contains(target))
435                             {
436                                 if (_useLinkedList)
437                                 {
438                                     link = ExternalSourcesTreeMethods.Search(source, target);
439                                 }
440                                 else
441                                 {
442                                     link = InternalSourcesTreeMethods.Search(source, target);
443                                 }
444                             }
445                             else
446                             {
447                                 if (_useLinkedList ||
                                    ↪  GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
                                    ↪  InternalTargetsTreeMethods.CountUsages(target)))
448                                 {
449                                     link = InternalTargetsTreeMethods.Search(source, target);
450                                 }
451                                 else
452                                 {
453                                     link = InternalSourcesTreeMethods.Search(source, target);
454                                 }
455                             }
456                         }
457                         else
458                         {
459                             if (_useLinkedList ||
                                ↪  GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
                                ↪  InternalTargetsTreeMethods.CountUsages(target)))
460                             {
461                                 link = InternalTargetsTreeMethods.Search(source, target);
462                             }
463                             else
464                             {
465                                 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(TLinkAddress);
                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.");
    }

    /// <summary>
    /// <para>
    /// Eaches the handler.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <param name="handler">
    /// <para>The handler.</para>
    /// <para></para>
    /// </param>
    /// <param name="restriction">
    /// <para>The substitution.</para>
    /// <para></para>
    /// </param>
    /// <exception cref="NotSupportedException">
    /// <para>Другие размеры и способы ограничений не поддерживаются.</para>
    /// <para></para>
    /// </exception>
    /// <returns>
    /// <para>The link</para>
    /// <para></para>
    /// </returns>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public virtual TLinkAddress Each(IList<TLinkAddress>? restriction,
    ↪  ReadHandler<TLinkAddress>? handler)
    {
        var constants = Constants;
        var @break = constants.Break;
        if (restriction.Count == 0)
        {
            for (var link = GetOne(); LessOrEqualThan(link,
            ↪  GetHeaderReference().AllocatedLinks); link = Increment(link))
```

```
539                 {
540                     if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
541                     {
542                         return @break;
543                     }
544                 }
545                 return @break;
546             }
547             var @continue = constants.Continue;
548             var any = constants.Any;
549             var index = restriction[constants.IndexPart];
550             if (restriction.Count == 1)
551             {
552                 if (AreEqual(index, any))
553                 {
554                     return Each(Array.Empty<TLinkAddress>(), handler);
555                 }
556                 if (!Exists(index))
557                 {
558                     return @continue;
559                 }
560                 return handler(GetLinkStruct(index));
561             }
562             if (restriction.Count == 2)
563             {
564                 var value = restriction[1];
565                 if (AreEqual(index, any))
566                 {
567                     if (AreEqual(value, any))
568                     {
569                         return Each(Array.Empty<TLinkAddress>(), handler);
570                     }
571                     if (AreEqual(Each(new Link<TLinkAddress>(index, value, any), handler),
                        ↪ @break))
572                     {
573                         return @break;
574                     }
575                     return Each(new Link<TLinkAddress>(index, any, value), handler);
576                 }
577                 else
578                 {
579                     if (!Exists(index))
580                     {
581                         return @continue;
582                     }
583                     if (AreEqual(value, any))
584                     {
585                         return handler(GetLinkStruct(index));
586                     }
587                     ref var storedLinkValue = ref GetLinkDataPartReference(index);
588                     if (AreEqual(storedLinkValue.Source, value) ||
589                         AreEqual(storedLinkValue.Target, value))
590                     {
591                         return handler(GetLinkStruct(index));
592                     }
593                     return @continue;
594                 }
595             }
596             if (restriction.Count == 3)
597             {
598                 var externalReferencesRange = constants.ExternalReferencesRange;
599                 var source = restriction[constants.SourcePart];
600                 var target = restriction[constants.TargetPart];
601                 if (AreEqual(index, any))
602                 {
603                     if (AreEqual(source, any) && AreEqual(target, any))
604                     {
605                         return Each(Array.Empty<TLinkAddress>(), handler);
606                     }
607                     else if (AreEqual(source, any))
608                     {
609                         if (externalReferencesRange.HasValue &&
                            ↪ externalReferencesRange.Value.Contains(target))
610                         {
611                             return ExternalTargetsTreeMethods.EachUsage(target, handler);
612                         }
613                         else
614                         {
```

```
615                         return InternalTargetsTreeMethods.EachUsage(target, handler);
616                     }
617                 }
618                 else if (AreEqual(target, any))
619                 {
620                     if (externalReferencesRange.HasValue &&
                        ↪  externalReferencesRange.Value.Contains(source))
621                     {
622                         return ExternalSourcesTreeMethods.EachUsage(source, handler);
623                     }
624                     else
625                     {
626                         if (_useLinkedList)
627                         {
628                             return InternalSourcesListMethods.EachUsage(source, handler);
629                         }
630                         else
631                         {
632                             return InternalSourcesTreeMethods.EachUsage(source, handler);
633                         }
634                     }
635                 }
636                 else //if(source != Any && target != Any)
637                 {
638                     TLinkAddress link;
639                     if (externalReferencesRange.HasValue)
640                     {
641                         if (externalReferencesRange.Value.Contains(source) &&
                            ↪  externalReferencesRange.Value.Contains(target))
642                         {
643                             link = ExternalSourcesTreeMethods.Search(source, target);
644                         }
645                         else if (externalReferencesRange.Value.Contains(source))
646                         {
647                             link = InternalTargetsTreeMethods.Search(source, target);
648                         }
649                         else if (externalReferencesRange.Value.Contains(target))
650                         {
651                             if (_useLinkedList)
652                             {
653                                 link = ExternalSourcesTreeMethods.Search(source, target);
654                             }
655                             else
656                             {
657                                 link = InternalSourcesTreeMethods.Search(source, target);
658                             }
659                         }
660                         else
661                         {
662                             if (_useLinkedList ||
                                ↪  GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
                                ↪  InternalTargetsTreeMethods.CountUsages(target)))
663                             {
664                                 link = InternalTargetsTreeMethods.Search(source, target);
665                             }
666                             else
667                             {
668                                 link = InternalSourcesTreeMethods.Search(source, target);
669                             }
670                         }
671                     }
672                     else
673                     {
674                         if (_useLinkedList ||
                            ↪  GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
                            ↪  InternalTargetsTreeMethods.CountUsages(target)))
675                         {
676                             link = InternalTargetsTreeMethods.Search(source, target);
677                         }
678                         else
679                         {
680                             link = InternalSourcesTreeMethods.Search(source, target);
681                         }
682                     }
683                     return AreEqual(link, constants.Null) ? @continue :
                        ↪  handler(GetLinkStruct(link));
684                 }
685             }
```

```
                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(TLinkAddress);
                    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 TLinkAddress Update(IList<TLinkAddress>? restriction,
        ↪  IList<TLinkAddress>? substitution, WriteHandler<TLinkAddress>? handler)
        {
            var constants = Constants;
            var @null = constants.Null;
            var externalReferencesRange = constants.ExternalReferencesRange;
            var linkIndex = restriction[constants.IndexPart];
            var before = GetLinkStruct(linkIndex);
            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);
```

```
759                     }
760                 }
761             }
762             if (!AreEqual(target, @null))
763             {
764                 if (externalReferencesRange.HasValue &&
                    ↪ externalReferencesRange.Value.Contains(target))
765                 {
766                     ExternalTargetsTreeMethods.Detach(ref rootAsTarget, linkIndex);
767                 }
768                 else
769                 {
770                     InternalTargetsTreeMethods.Detach(ref
                        ↪ GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
771                 }
772             }
773             source = link.Source = substitution[constants.SourcePart];
774             target = link.Target = substitution[constants.TargetPart];
775             if (!AreEqual(source, @null))
776             {
777                 if (externalReferencesRange.HasValue &&
                    ↪ externalReferencesRange.Value.Contains(source))
778                 {
779                     ExternalSourcesTreeMethods.Attach(ref rootAsSource, linkIndex);
780                 }
781                 else
782                 {
783                     if (_useLinkedList)
784                     {
785                         InternalSourcesListMethods.AttachAsLast(source, linkIndex);
786                     }
787                     else
788                     {
789                         InternalSourcesTreeMethods.Attach(ref
                            ↪ GetLinkIndexPartReference(source).RootAsSource, linkIndex);
790                     }
791                 }
792             }
793             if (!AreEqual(target, @null))
794             {
795                 if (externalReferencesRange.HasValue &&
                    ↪ externalReferencesRange.Value.Contains(target))
796                 {
797                     ExternalTargetsTreeMethods.Attach(ref rootAsTarget, linkIndex);
798                 }
799                 else
800                 {
801                     InternalTargetsTreeMethods.Attach(ref
                        ↪ GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
802                 }
803             }
804             return handler?.Invoke(before, new Link<TLinkAddress>(linkIndex, source, target)) ??
                ↪ Constants.Continue;
805         }
806 
807         /// <remarks>
808         /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
                ↪ пространство
809         /// </remarks>
810         [MethodImpl(MethodImplOptions.AggressiveInlining)]
811         public virtual TLinkAddress Create(IList<TLinkAddress>? substitution,
                ↪ WriteHandler<TLinkAddress>? handler)
812         {
813             ref var header = ref GetHeaderReference();
814             var freeLink = header.FirstFreeLink;
815             if (!AreEqual(freeLink, Constants.Null))
816             {
817                 UnusedLinksListMethods.Detach(freeLink);
818             }
819             else
820             {
821                 var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
822                 if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
823                 {
824                     throw new
                        ↪ LinksLimitReachedException<TLinkAddress>(maximumPossibleInnerReference);
825                 }
826                 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 handler?.Invoke(null, GetLinkStruct(freeLink)) ?? Constants.Continue;
        }

        /// <summary>
        /// <para>
        /// Deletes the substitution.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restriction">
        /// <para>The substitution.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLinkAddress Delete(IList<TLinkAddress>? restriction,
        ↪  WriteHandler<TLinkAddress>? handler)
        {
            ref var header = ref GetHeaderReference();
            var link = restriction[Constants.IndexPart];
            var before = GetLinkStruct(link);
            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;
                }
            }
            return handler?.Invoke(before, null) ?? Constants.Continue;
        }

        /// <summary>
        /// <para>
        /// Gets the link struct using the specified link index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="linkIndex">
        /// <para>The link index.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A list of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IList<TLinkAddress>? GetLinkStruct(TLinkAddress linkIndex)
        {
            ref var link = ref GetLinkDataPartReference(linkIndex);
            return new Link<TLinkAddress>(linkIndex, link.Source, link.Target);
        }

        /// <remarks>
```

```
901             /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
                ↪  адрес реально поменялся
902             ///
903             /// Указатель this.links может быть в том же месте,
904             /// так как 0-я связь не используется и имеет такой же размер как Header,
905             /// поэтому header размещается в том же месте, что и 0-я связь
906             /// </remarks>
907             [MethodImpl(MethodImplOptions.AggressiveInlining)]
908             protected abstract void SetPointers(IResizableDirectMemory dataMemory,
                ↪  IResizableDirectMemory indexMemory);
909
910             /// <summary>
911             /// <para>
912             /// Resets the pointers.
913             /// </para>
914             /// <para></para>
915             /// </summary>
916             [MethodImpl(MethodImplOptions.AggressiveInlining)]
917             protected virtual void ResetPointers()
918             {
919                 InternalSourcesListMethods = null;
920                 InternalSourcesTreeMethods = null;
921                 ExternalSourcesTreeMethods = null;
922                 InternalTargetsTreeMethods = null;
923                 ExternalTargetsTreeMethods = null;
924                 UnusedLinksListMethods = null;
925             }
926
927             /// <summary>
928             /// <para>
929             /// Gets the header reference.
930             /// </para>
931             /// <para></para>
932             /// </summary>
933             /// <returns>
934             /// <para>A ref links header of t link</para>
935             /// <para></para>
936             /// </returns>
937             [MethodImpl(MethodImplOptions.AggressiveInlining)]
938             protected abstract ref LinksHeader<TLinkAddress> GetHeaderReference();
939
940             /// <summary>
941             /// <para>
942             /// Gets the link data part reference using the specified link index.
943             /// </para>
944             /// <para></para>
945             /// </summary>
946             /// <param name="linkIndex">
947             /// <para>The link index.</para>
948             /// <para></para>
949             /// </param>
950             /// <returns>
951             /// <para>A ref raw link data part of t link</para>
952             /// <para></para>
953             /// </returns>
954             [MethodImpl(MethodImplOptions.AggressiveInlining)]
955             protected abstract ref RawLinkDataPart<TLinkAddress>
                ↪  GetLinkDataPartReference(TLinkAddress linkIndex);
956
957             /// <summary>
958             /// <para>
959             /// Gets the link index part reference using the specified link index.
960             /// </para>
961             /// <para></para>
962             /// </summary>
963             /// <param name="linkIndex">
964             /// <para>The link index.</para>
965             /// <para></para>
966             /// </param>
967             /// <returns>
968             /// <para>A ref raw link index part of t link</para>
969             /// <para></para>
970             /// </returns>
971             [MethodImpl(MethodImplOptions.AggressiveInlining)]
972             protected abstract ref RawLinkIndexPart<TLinkAddress>
                ↪  GetLinkIndexPartReference(TLinkAddress linkIndex);
973
974             /// <summary>
```

```
        /// <para>
        /// Determines whether this instance exists.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool Exists(TLinkAddress link)
            => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
            && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
            && !IsUnusedLink(link);

        /// <summary>
        /// <para>
        /// Determines whether this instance is unused link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="linkIndex">
        /// <para>The link index.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool IsUnusedLink(TLinkAddress 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;
            }
        }

        /// <summary>
        /// <para>
        /// Gets the one.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLinkAddress GetOne() => _one;

        /// <summary>
        /// <para>
        /// Gets the zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLinkAddress GetZero() => default;

```

```
1050            /// <summary>
1051            /// <para>
1052            /// Determines whether this instance are equal.
1053            /// </para>
1054            /// <para></para>
1055            /// </summary>
1056            /// <param name="first">
1057            /// <para>The first.</para>
1058            /// <para></para>
1059            /// </param>
1060            /// <param name="second">
1061            /// <para>The second.</para>
1062            /// <para></para>
1063            /// </param>
1064            /// <returns>
1065            /// <para>The bool</para>
1066            /// <para></para>
1067            /// </returns>
1068            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1069            protected virtual bool AreEqual(TLinkAddress first, TLinkAddress second) =>
                ↪ _equalityComparer.Equals(first, second);
1070
1071            /// <summary>
1072            /// <para>
1073            /// Determines whether this instance less than.
1074            /// </para>
1075            /// <para></para>
1076            /// </summary>
1077            /// <param name="first">
1078            /// <para>The first.</para>
1079            /// <para></para>
1080            /// </param>
1081            /// <param name="second">
1082            /// <para>The second.</para>
1083            /// <para></para>
1084            /// </param>
1085            /// <returns>
1086            /// <para>The bool</para>
1087            /// <para></para>
1088            /// </returns>
1089            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1090            protected virtual bool LessThan(TLinkAddress first, TLinkAddress second) =>
                ↪ _comparer.Compare(first, second) < 0;
1091
1092            /// <summary>
1093            /// <para>
1094            /// Determines whether this instance less or equal than.
1095            /// </para>
1096            /// <para></para>
1097            /// </summary>
1098            /// <param name="first">
1099            /// <para>The first.</para>
1100            /// <para></para>
1101            /// </param>
1102            /// <param name="second">
1103            /// <para>The second.</para>
1104            /// <para></para>
1105            /// </param>
1106            /// <returns>
1107            /// <para>The bool</para>
1108            /// <para></para>
1109            /// </returns>
1110            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1111            protected virtual bool LessOrEqualThan(TLinkAddress first, TLinkAddress second) =>
                ↪ _comparer.Compare(first, second) <= 0;
1112
1113            /// <summary>
1114            /// <para>
1115            /// Determines whether this instance greater than.
1116            /// </para>
1117            /// <para></para>
1118            /// </summary>
1119            /// <param name="first">
1120            /// <para>The first.</para>
1121            /// <para></para>
1122            /// </param>
1123            /// <param name="second">
1124            /// <para>The second.</para>
```

```
1125            /// <para></para>
1126            /// </param>
1127            /// <returns>
1128            /// <para>The bool</para>
1129            /// <para></para>
1130            /// </returns>
1131            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1132            protected virtual bool GreaterThan(TLinkAddress first, TLinkAddress second) =>
                ↪  _comparer.Compare(first, second) > 0;
1133
1134            /// <summary>
1135            /// <para>
1136            /// Determines whether this instance greater or equal than.
1137            /// </para>
1138            /// <para></para>
1139            /// </summary>
1140            /// <param name="first">
1141            /// <para>The first.</para>
1142            /// <para></para>
1143            /// </param>
1144            /// <param name="second">
1145            /// <para>The second.</para>
1146            /// <para></para>
1147            /// </param>
1148            /// <returns>
1149            /// <para>The bool</para>
1150            /// <para></para>
1151            /// </returns>
1152            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1153            protected virtual bool GreaterOrEqualThan(TLinkAddress first, TLinkAddress second) =>
                ↪  _comparer.Compare(first, second) >= 0;
1154
1155            /// <summary>
1156            /// <para>
1157            /// Converts the to int 64 using the specified value.
1158            /// </para>
1159            /// <para></para>
1160            /// </summary>
1161            /// <param name="value">
1162            /// <para>The value.</para>
1163            /// <para></para>
1164            /// </param>
1165            /// <returns>
1166            /// <para>The long</para>
1167            /// <para></para>
1168            /// </returns>
1169            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1170            protected virtual long ConvertToInt64(TLinkAddress value) =>
                ↪  _addressToInt64Converter.Convert(value);
1171
1172            /// <summary>
1173            /// <para>
1174            /// Converts the to address using the specified value.
1175            /// </para>
1176            /// <para></para>
1177            /// </summary>
1178            /// <param name="value">
1179            /// <para>The value.</para>
1180            /// <para></para>
1181            /// </param>
1182            /// <returns>
1183            /// <para>The link</para>
1184            /// <para></para>
1185            /// </returns>
1186            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1187            protected virtual TLinkAddress ConvertToAddress(long value) =>
                ↪  _int64ToAddressConverter.Convert(value);
1188
1189            /// <summary>
1190            /// <para>
1191            /// Adds the first.
1192            /// </para>
1193            /// <para></para>
1194            /// </summary>
1195            /// <param name="first">
1196            /// <para>The first.</para>
1197            /// <para></para>
1198            /// </param>
```

```
1199            /// <param name="second">
1200            /// <para>The second.</para>
1201            /// <para></para>
1202            /// </param>
1203            /// <returns>
1204            /// <para>The link</para>
1205            /// <para></para>
1206            /// </returns>
1207            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1208            protected virtual TLinkAddress Add(TLinkAddress first, TLinkAddress second) =>
                ↪ Arithmetic<TLinkAddress>.Add(first, second);
1209
1210            /// <summary>
1211            /// <para>
1212            /// Subtracts the first.
1213            /// </para>
1214            /// <para></para>
1215            /// </summary>
1216            /// <param name="first">
1217            /// <para>The first.</para>
1218            /// <para></para>
1219            /// </param>
1220            /// <param name="second">
1221            /// <para>The second.</para>
1222            /// <para></para>
1223            /// </param>
1224            /// <returns>
1225            /// <para>The link</para>
1226            /// <para></para>
1227            /// </returns>
1228            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1229            protected virtual TLinkAddress Subtract(TLinkAddress first, TLinkAddress second) =>
                ↪ Arithmetic<TLinkAddress>.Subtract(first, second);
1230
1231            /// <summary>
1232            /// <para>
1233            /// Increments the link.
1234            /// </para>
1235            /// <para></para>
1236            /// </summary>
1237            /// <param name="link">
1238            /// <para>The link.</para>
1239            /// <para></para>
1240            /// </param>
1241            /// <returns>
1242            /// <para>The link</para>
1243            /// <para></para>
1244            /// </returns>
1245            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1246            protected virtual TLinkAddress Increment(TLinkAddress link) =>
                ↪ Arithmetic<TLinkAddress>.Increment(link);
1247
1248            /// <summary>
1249            /// <para>
1250            /// Decrements the link.
1251            /// </para>
1252            /// <para></para>
1253            /// </summary>
1254            /// <param name="link">
1255            /// <para>The link.</para>
1256            /// <para></para>
1257            /// </param>
1258            /// <returns>
1259            /// <para>The link</para>
1260            /// <para></para>
1261            /// </returns>
1262            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1263            protected virtual TLinkAddress Decrement(TLinkAddress link) =>
                ↪ Arithmetic<TLinkAddress>.Decrement(link);
1264
1265            #region Disposable
1266
1267            /// <summary>
1268            /// <para>
1269            /// Gets the allow multiple dispose calls value.
1270            /// </para>
1271            /// <para></para>
1272            /// </summary>
```

```
1273          protected override bool AllowMultipleDisposeCalls
1274          {
1275              [MethodImpl(MethodImplOptions.AggressiveInlining)]
1276              get => true;
1277          }
1278
1279          /// <summary>
1280          /// <para>
1281          /// Disposes the manual.
1282          /// </para>
1283          /// <para></para>
1284          /// </summary>
1285          /// <param name="manual">
1286          /// <para>The manual.</para>
1287          /// <para></para>
1288          /// </param>
1289          /// <param name="wasDisposed">
1290          /// <para>The was disposed.</para>
1291          /// <para></para>
1292          /// </param>
1293          [MethodImpl(MethodImplOptions.AggressiveInlining)]
1294          protected override void Dispose(bool manual, bool wasDisposed)
1295          {
1296              if (!wasDisposed)
1297              {
1298                  ResetPointers();
1299                  _dataMemory.DisposeIfPossible();
1300                  _indexMemory.DisposeIfPossible();
1301              }
1302          }
1303
1304          #endregion
1305      }
1306  }
```

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

```
1   using System.Runtime.CompilerServices;
2   using Platform.Collections.Methods.Lists;
3   using Platform.Converters;
4   using static System.Runtime.CompilerServices.Unsafe;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Data.Doublets.Memory.Split.Generic
9   {
10      /// <summary>
11      /// <para>
12      /// Represents the unused links list methods.
13      /// </para>
14      /// <para></para>
15      /// </summary>
16      /// <seealso cref="AbsoluteCircularDoublyLinkedListMethods{TLinkAddress}"/>
17      /// <seealso cref="ILinksListMethods{TLinkAddress}"/>
18      public unsafe class UnusedLinksListMethods<TLinkAddress> :
        ↪  AbsoluteCircularDoublyLinkedListMethods<TLinkAddress>, ILinksListMethods<TLinkAddress>
19      {
20          private static readonly UncheckedConverter<TLinkAddress, long> _addressToInt64Converter
            ↪  = UncheckedConverter<TLinkAddress, long>.Default;
21          private readonly byte* _links;
22          private readonly byte* _header;
23
24          /// <summary>
25          /// <para>
26          /// Initializes a new <see cref="UnusedLinksListMethods"/> instance.
27          /// </para>
28          /// <para></para>
29          /// </summary>
30          /// <param name="links">
31          /// <para>A links.</para>
32          /// <para></para>
33          /// </param>
34          /// <param name="header">
35          /// <para>A header.</para>
36          /// <para></para>
37          /// </param>
38          [MethodImpl(MethodImplOptions.AggressiveInlining)]
39          public UnusedLinksListMethods(byte* links, byte* header)
40          {
41              _links = links;
```

```
            _header = header;
        }

        /// <summary>
        /// <para>
        /// Gets the header reference.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A ref links header of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref LinksHeader<TLinkAddress> GetHeaderReference() => ref
        ↪  AsRef<LinksHeader<TLinkAddress>>(_header);

        /// <summary>
        /// <para>
        /// Gets the link data part reference using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link data part of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref RawLinkDataPart<TLinkAddress>
        ↪  GetLinkDataPartReference(TLinkAddress link) => ref
        ↪  AsRef<RawLinkDataPart<TLinkAddress>>(_links +
        ↪  (RawLinkDataPart<TLinkAddress>.SizeInBytes *
        ↪  _addressToInt64Converter.Convert(link)));

        /// <summary>
        /// <para>
        /// Gets the first.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetFirst() => GetHeaderReference().FirstFreeLink;

        /// <summary>
        /// <para>
        /// Gets the last.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetLast() => GetHeaderReference().LastFreeLink;

        /// <summary>
        /// <para>
        /// Gets the previous using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
```

```
115             [MethodImpl(MethodImplOptions.AggressiveInlining)]
116             protected override TLinkAddress GetPrevious(TLinkAddress element) =>
                ↪ GetLinkDataPartReference(element).Source;
117
118             /// <summary>
119             /// <para>
120             /// Gets the next using the specified element.
121             /// </para>
122             /// <para></para>
123             /// </summary>
124             /// <param name="element">
125             /// <para>The element.</para>
126             /// <para></para>
127             /// </param>
128             /// <returns>
129             /// <para>The link</para>
130             /// <para></para>
131             /// </returns>
132             [MethodImpl(MethodImplOptions.AggressiveInlining)]
133             protected override TLinkAddress GetNext(TLinkAddress element) =>
                ↪ GetLinkDataPartReference(element).Target;
134
135             /// <summary>
136             /// <para>
137             /// Gets the size.
138             /// </para>
139             /// <para></para>
140             /// </summary>
141             /// <returns>
142             /// <para>The link</para>
143             /// <para></para>
144             /// </returns>
145             [MethodImpl(MethodImplOptions.AggressiveInlining)]
146             protected override TLinkAddress GetSize() => GetHeaderReference().FreeLinks;
147
148             /// <summary>
149             /// <para>
150             /// Sets the first using the specified element.
151             /// </para>
152             /// <para></para>
153             /// </summary>
154             /// <param name="element">
155             /// <para>The element.</para>
156             /// <para></para>
157             /// </param>
158             [MethodImpl(MethodImplOptions.AggressiveInlining)]
159             protected override void SetFirst(TLinkAddress element) =>
                ↪ GetHeaderReference().FirstFreeLink = element;
160
161             /// <summary>
162             /// <para>
163             /// Sets the last using the specified element.
164             /// </para>
165             /// <para></para>
166             /// </summary>
167             /// <param name="element">
168             /// <para>The element.</para>
169             /// <para></para>
170             /// </param>
171             [MethodImpl(MethodImplOptions.AggressiveInlining)]
172             protected override void SetLast(TLinkAddress element) =>
                ↪ GetHeaderReference().LastFreeLink = element;
173
174             /// <summary>
175             /// <para>
176             /// Sets the previous using the specified element.
177             /// </para>
178             /// <para></para>
179             /// </summary>
180             /// <param name="element">
181             /// <para>The element.</para>
182             /// <para></para>
183             /// </param>
184             /// <param name="previous">
185             /// <para>The previous.</para>
186             /// <para></para>
187             /// </param>
188             [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
189         protected override void SetPrevious(TLinkAddress element, TLinkAddress previous) =>
            ↪ GetLinkDataPartReference(element).Source = previous;
190
191         /// <summary>
192         /// <para>
193         /// Sets the next using the specified element.
194         /// </para>
195         /// <para></para>
196         /// </summary>
197         /// <param name="element">
198         /// <para>The element.</para>
199         /// <para></para>
200         /// </param>
201         /// <param name="next">
202         /// <para>The next.</para>
203         /// <para></para>
204         /// </param>
205         [MethodImpl(MethodImplOptions.AggressiveInlining)]
206         protected override void SetNext(TLinkAddress element, TLinkAddress next) =>
            ↪ GetLinkDataPartReference(element).Target = next;
207
208         /// <summary>
209         /// <para>
210         /// Sets the size using the specified size.
211         /// </para>
212         /// <para></para>
213         /// </summary>
214         /// <param name="size">
215         /// <para>The size.</para>
216         /// <para></para>
217         /// </param>
218         [MethodImpl(MethodImplOptions.AggressiveInlining)]
219         protected override void SetSize(TLinkAddress size) => GetHeaderReference().FreeLinks =
            ↪ size;
220     }
221 }
```

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

```
1  using Platform.Unsafe;
2  using System;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Memory.Split
9  {
10     /// <summary>
11     /// <para>
12     /// The raw link data part.
13     /// </para>
14     /// <para></para>
15     /// </summary>
16     public struct RawLinkDataPart<TLinkAddress> : IEquatable<RawLinkDataPart<TLinkAddress>>
17     {
18         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
            ↪ EqualityComparer<TLinkAddress>.Default;
19
20         /// <summary>
21         /// <para>
22         /// The size.
23         /// </para>
24         /// <para></para>
25         /// </summary>
26         public static readonly long SizeInBytes = Structure<RawLinkDataPart<TLinkAddress>>.Size;
27
28         /// <summary>
29         /// <para>
30         /// The source.
31         /// </para>
32         /// <para></para>
33         /// </summary>
34         public TLinkAddress Source;
35         /// <summary>
36         /// <para>
37         /// The target.
38         /// </para>
39         /// <para></para>
40         /// </summary>
```

```
41          public TLinkAddress Target;
42
43          /// <summary>
44          /// <para>
45          /// Determines whether this instance equals.
46          /// </para>
47          /// <para></para>
48          /// </summary>
49          /// <param name="obj">
50          /// <para>The obj.</para>
51          /// <para></para>
52          /// </param>
53          /// <returns>
54          /// <para>The bool</para>
55          /// <para></para>
56          /// </returns>
57          [MethodImpl(MethodImplOptions.AggressiveInlining)]
58          public override bool Equals(object obj) => obj is RawLinkDataPart<TLinkAddress> link ?
            ↪  Equals(link) : false;
59
60          /// <summary>
61          /// <para>
62          /// Determines whether this instance equals.
63          /// </para>
64          /// <para></para>
65          /// </summary>
66          /// <param name="other">
67          /// <para>The other.</para>
68          /// <para></para>
69          /// </param>
70          /// <returns>
71          /// <para>The bool</para>
72          /// <para></para>
73          /// </returns>
74          [MethodImpl(MethodImplOptions.AggressiveInlining)]
75          public bool Equals(RawLinkDataPart<TLinkAddress> other)
76              => _equalityComparer.Equals(Source, other.Source)
77              && _equalityComparer.Equals(Target, other.Target);
78
79          /// <summary>
80          /// <para>
81          /// Gets the hash code.
82          /// </para>
83          /// <para></para>
84          /// </summary>
85          /// <returns>
86          /// <para>The int</para>
87          /// <para></para>
88          /// </returns>
89          [MethodImpl(MethodImplOptions.AggressiveInlining)]
90          public override int GetHashCode() => (Source, Target).GetHashCode();
91
92          [MethodImpl(MethodImplOptions.AggressiveInlining)]
93          public static bool operator ==(RawLinkDataPart<TLinkAddress> left,
            ↪  RawLinkDataPart<TLinkAddress> right) => left.Equals(right);
94
95          [MethodImpl(MethodImplOptions.AggressiveInlining)]
96          public static bool operator !=(RawLinkDataPart<TLinkAddress> left,
            ↪  RawLinkDataPart<TLinkAddress> right) => !(left == right);
97      }
98  }
```

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

```
1   using Platform.Unsafe;
2   using System;
3   using System.Collections.Generic;
4   using System.Runtime.CompilerServices;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Data.Doublets.Memory.Split
9   {
10      /// <summary>
11      /// <para>
12      /// The raw link index part.
13      /// </para>
14      /// <para></para>
15      /// </summary>
```

```
16      public struct RawLinkIndexPart<TLinkAddress> : IEquatable<RawLinkIndexPart<TLinkAddress>>
17      {
18          private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
            ↪  EqualityComparer<TLinkAddress>.Default;
19
20          /// <summary>
21          /// <para>
22          /// The size.
23          /// </para>
24          /// <para></para>
25          /// </summary>
26          public static readonly long SizeInBytes = Structure<RawLinkIndexPart<TLinkAddress>>.Size;
27
28          /// <summary>
29          /// <para>
30          /// The root as source.
31          /// </para>
32          /// <para></para>
33          /// </summary>
34          public TLinkAddress RootAsSource;
35          /// <summary>
36          /// <para>
37          /// The left as source.
38          /// </para>
39          /// <para></para>
40          /// </summary>
41          public TLinkAddress LeftAsSource;
42          /// <summary>
43          /// <para>
44          /// The right as source.
45          /// </para>
46          /// <para></para>
47          /// </summary>
48          public TLinkAddress RightAsSource;
49          /// <summary>
50          /// <para>
51          /// The size as source.
52          /// </para>
53          /// <para></para>
54          /// </summary>
55          public TLinkAddress SizeAsSource;
56          /// <summary>
57          /// <para>
58          /// The root as target.
59          /// </para>
60          /// <para></para>
61          /// </summary>
62          public TLinkAddress RootAsTarget;
63          /// <summary>
64          /// <para>
65          /// The left as target.
66          /// </para>
67          /// <para></para>
68          /// </summary>
69          public TLinkAddress LeftAsTarget;
70          /// <summary>
71          /// <para>
72          /// The right as target.
73          /// </para>
74          /// <para></para>
75          /// </summary>
76          public TLinkAddress RightAsTarget;
77          /// <summary>
78          /// <para>
79          /// The size as target.
80          /// </para>
81          /// <para></para>
82          /// </summary>
83          public TLinkAddress SizeAsTarget;
84
85          /// <summary>
86          /// <para>
87          /// Determines whether this instance equals.
88          /// </para>
89          /// <para></para>
90          /// </summary>
91          /// <param name="obj">
92          /// <para>The obj.</para>
93          /// <para></para>
```

```
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Equals(object obj) => obj is RawLinkIndexPart<TLinkAddress> link ?
        ↪  Equals(link) : false;

        /// <summary>
        /// <para>
        /// Determines whether this instance equals.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="other">
        /// <para>The other.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Equals(RawLinkIndexPart<TLinkAddress> 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);

        /// <summary>
        /// <para>
        /// Gets the hash code.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The int</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override int GetHashCode() => (RootAsSource, LeftAsSource, RightAsSource,
        ↪  SizeAsSource, RootAsTarget, LeftAsTarget, RightAsTarget, SizeAsTarget).GetHashCode();

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

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

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

```
using System.Runtime.CompilerServices;
using Platform.Data.Doublets.Memory.Split.Generic;
using TLinkAddress = System.UInt32;

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    /// <summary>
    /// <para>
    /// Represents the int 32 external links recursionless size balanced tree methods base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ExternalLinksRecursionlessSizeBalancedTreeMethodsBase{TLinkAddress}"/>
    /// <seealso cref="ILinksTreeMethods{TLinkAddress}"/>
    public unsafe abstract class UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase :
    ↪  ExternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLinkAddress>,
    ↪  ILinksTreeMethods<TLinkAddress>
```

```
18      {
19          /// <summary>
20          /// <para>
21          /// The links data parts.
22          /// </para>
23          /// <para></para>
24          /// </summary>
25          protected new readonly RawLinkDataPart<TLinkAddress>* LinksDataParts;
26          /// <summary>
27          /// <para>
28          /// The links index parts.
29          /// </para>
30          /// <para></para>
31          /// </summary>
32          protected new readonly RawLinkIndexPart<TLinkAddress>* LinksIndexParts;
33          /// <summary>
34          /// <para>
35          /// The header.
36          /// </para>
37          /// <para></para>
38          /// </summary>
39          protected new readonly LinksHeader<TLinkAddress>* Header;
40
41          /// <summary>
42          /// <para>
43          /// Initializes a new <see
            ↪  cref="UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase"/> instance.
44          /// </para>
45          /// <para></para>
46          /// </summary>
47          /// <param name="constants">
48          /// <para>A constants.</para>
49          /// <para></para>
50          /// </param>
51          /// <param name="linksDataParts">
52          /// <para>A links data parts.</para>
53          /// <para></para>
54          /// </param>
55          /// <param name="linksIndexParts">
56          /// <para>A links index parts.</para>
57          /// <para></para>
58          /// </param>
59          /// <param name="header">
60          /// <para>A header.</para>
61          /// <para></para>
62          /// </param>
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          protected UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLi
            ↪  nkAddress> constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
            ↪  RawLinkIndexPart<TLinkAddress>* linksIndexParts, LinksHeader<TLinkAddress>* header)
65              : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
66          {
67              LinksDataParts = linksDataParts;
68              LinksIndexParts = linksIndexParts;
69              Header = header;
70          }
71
72          /// <summary>
73          /// <para>
74          /// Gets the zero.
75          /// </para>
76          /// <para></para>
77          /// </summary>
78          /// <returns>
79          /// <para>The link</para>
80          /// <para></para>
81          /// </returns>
82          [MethodImpl(MethodImplOptions.AggressiveInlining)]
83          protected override TLinkAddress GetZero() => 0U;
84
85          /// <summary>
86          /// <para>
87          /// Determines whether this instance equal to zero.
88          /// </para>
89          /// <para></para>
90          /// </summary>
91          /// <param name="value">
92          /// <para>The value.</para>
```

```
93          /// <para></para>
94          /// </param>
95          /// <returns>
96          /// <para>The bool</para>
97          /// <para></para>
98          /// </returns>
99          [MethodImpl(MethodImplOptions.AggressiveInlining)]
100         protected override bool EqualToZero(TLinkAddress value) => value == 0U;
101
102         /// <summary>
103         /// <para>
104         /// Determines whether this instance are equal.
105         /// </para>
106         /// <para></para>
107         /// </summary>
108         /// <param name="first">
109         /// <para>The first.</para>
110         /// <para></para>
111         /// </param>
112         /// <param name="second">
113         /// <para>The second.</para>
114         /// <para></para>
115         /// </param>
116         /// <returns>
117         /// <para>The bool</para>
118         /// <para></para>
119         /// </returns>
120         [MethodImpl(MethodImplOptions.AggressiveInlining)]
121         protected override bool AreEqual(TLinkAddress first, TLinkAddress second) => first ==
            ↪ second;
122
123         /// <summary>
124         /// <para>
125         /// Determines whether this instance greater than zero.
126         /// </para>
127         /// <para></para>
128         /// </summary>
129         /// <param name="value">
130         /// <para>The value.</para>
131         /// <para></para>
132         /// </param>
133         /// <returns>
134         /// <para>The bool</para>
135         /// <para></para>
136         /// </returns>
137         [MethodImpl(MethodImplOptions.AggressiveInlining)]
138         protected override bool GreaterThanZero(TLinkAddress value) => value > 0U;
139
140         /// <summary>
141         /// <para>
142         /// Determines whether this instance greater than.
143         /// </para>
144         /// <para></para>
145         /// </summary>
146         /// <param name="first">
147         /// <para>The first.</para>
148         /// <para></para>
149         /// </param>
150         /// <param name="second">
151         /// <para>The second.</para>
152         /// <para></para>
153         /// </param>
154         /// <returns>
155         /// <para>The bool</para>
156         /// <para></para>
157         /// </returns>
158         [MethodImpl(MethodImplOptions.AggressiveInlining)]
159         protected override bool GreaterThan(TLinkAddress first, TLinkAddress second) => first >
            ↪ second;
160
161         /// <summary>
162         /// <para>
163         /// Determines whether this instance greater or equal than.
164         /// </para>
165         /// <para></para>
166         /// </summary>
167         /// <param name="first">
168         /// <para>The first.</para>
```

```
169         /// <para></para>
170         /// </param>
171         /// <param name="second">
172         /// <para>The second.</para>
173         /// <para></para>
174         /// </param>
175         /// <returns>
176         /// <para>The bool</para>
177         /// <para></para>
178         /// </returns>
179         [MethodImpl(MethodImplOptions.AggressiveInlining)]
180         protected override bool GreaterOrEqualThan(TLinkAddress first, TLinkAddress second) =>
            ↪ first >= second;
181
182         /// <summary>
183         /// <para>
184         /// Determines whether this instance greater or equal than zero.
185         /// </para>
186         /// <para></para>
187         /// </summary>
188         /// <param name="value">
189         /// <para>The value.</para>
190         /// <para></para>
191         /// </param>
192         /// <returns>
193         /// <para>The bool</para>
194         /// <para></para>
195         /// </returns>
196         [MethodImpl(MethodImplOptions.AggressiveInlining)]
197         protected override bool GreaterOrEqualThanZero(TLinkAddress value) => true; // value >=
            ↪ 0 is always true for ulong
198
199         /// <summary>
200         /// <para>
201         /// Determines whether this instance less or equal than zero.
202         /// </para>
203         /// <para></para>
204         /// </summary>
205         /// <param name="value">
206         /// <para>The value.</para>
207         /// <para></para>
208         /// </param>
209         /// <returns>
210         /// <para>The bool</para>
211         /// <para></para>
212         /// </returns>
213         [MethodImpl(MethodImplOptions.AggressiveInlining)]
214         protected override bool LessOrEqualThanZero(TLinkAddress value) => value == OUL; //
            ↪ value is always >= 0 for ulong
215
216         /// <summary>
217         /// <para>
218         /// Determines whether this instance less or equal than.
219         /// </para>
220         /// <para></para>
221         /// </summary>
222         /// <param name="first">
223         /// <para>The first.</para>
224         /// <para></para>
225         /// </param>
226         /// <param name="second">
227         /// <para>The second.</para>
228         /// <para></para>
229         /// </param>
230         /// <returns>
231         /// <para>The bool</para>
232         /// <para></para>
233         /// </returns>
234         [MethodImpl(MethodImplOptions.AggressiveInlining)]
235         protected override bool LessOrEqualThan(TLinkAddress first, TLinkAddress second) =>
            ↪ first <= second;
236
237         /// <summary>
238         /// <para>
239         /// Determines whether this instance less than zero.
240         /// </para>
241         /// <para></para>
242         /// </summary>
```

```
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessThanZero(TLinkAddress value) => false; // value < 0 is
        ↪  always false for ulong

        /// <summary>
        /// <para>
        /// Determines whether this instance less than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessThan(TLinkAddress first, TLinkAddress second) => first <
        ↪  second;

        /// <summary>
        /// <para>
        /// Increments the value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress Increment(TLinkAddress value) => ++value;

        /// <summary>
        /// <para>
        /// Decrements the value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress Decrement(TLinkAddress value) => --value;

        /// <summary>
        /// <para>
        /// Adds the first.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
```

```
319             /// <param name="second">
320             /// <para>The second.</para>
321             /// <para></para>
322             /// </param>
323             /// <returns>
324             /// <para>The link</para>
325             /// <para></para>
326             /// </returns>
327             [MethodImpl(MethodImplOptions.AggressiveInlining)]
328             protected override TLinkAddress Add(TLinkAddress first, TLinkAddress second) => first +
                ↪ second;
329
330             /// <summary>
331             /// <para>
332             /// Subtracts the first.
333             /// </para>
334             /// <para></para>
335             /// </summary>
336             /// <param name="first">
337             /// <para>The first.</para>
338             /// <para></para>
339             /// </param>
340             /// <param name="second">
341             /// <para>The second.</para>
342             /// <para></para>
343             /// </param>
344             /// <returns>
345             /// <para>The link</para>
346             /// <para></para>
347             /// </returns>
348             [MethodImpl(MethodImplOptions.AggressiveInlining)]
349             protected override TLinkAddress Subtract(TLinkAddress first, TLinkAddress second) =>
                ↪ first - second;
350
351             /// <summary>
352             /// <para>
353             /// Gets the header reference.
354             /// </para>
355             /// <para></para>
356             /// </summary>
357             /// <returns>
358             /// <para>A ref links header of t link</para>
359             /// <para></para>
360             /// </returns>
361             [MethodImpl(MethodImplOptions.AggressiveInlining)]
362             protected override ref LinksHeader<TLinkAddress> GetHeaderReference() => ref *Header;
363
364             /// <summary>
365             /// <para>
366             /// Gets the link data part reference using the specified link.
367             /// </para>
368             /// <para></para>
369             /// </summary>
370             /// <param name="link">
371             /// <para>The link.</para>
372             /// <para></para>
373             /// </param>
374             /// <returns>
375             /// <para>A ref raw link data part of t link</para>
376             /// <para></para>
377             /// </returns>
378             [MethodImpl(MethodImplOptions.AggressiveInlining)]
379             protected override ref RawLinkDataPart<TLinkAddress>
                ↪ GetLinkDataPartReference(TLinkAddress link) => ref LinksDataParts[link];
380
381             /// <summary>
382             /// <para>
383             /// Gets the link index part reference using the specified link.
384             /// </para>
385             /// <para></para>
386             /// </summary>
387             /// <param name="link">
388             /// <para>The link.</para>
389             /// <para></para>
390             /// </param>
391             /// <returns>
392             /// <para>A ref raw link index part of t link</para>
393             /// <para></para>
```

```
394         /// </returns>
395         [MethodImpl(MethodImplOptions.AggressiveInlining)]
396         protected override ref RawLinkIndexPart<TLinkAddress>
            ↪ GetLinkIndexPartReference(TLinkAddress link) => ref LinksIndexParts[link];
397
398         /// <summary>
399         /// <para>
400         /// Determines whether this instance first is to the left of second.
401         /// </para>
402         /// <para></para>
403         /// </summary>
404         /// <param name="first">
405         /// <para>The first.</para>
406         /// <para></para>
407         /// </param>
408         /// <param name="second">
409         /// <para>The second.</para>
410         /// <para></para>
411         /// </param>
412         /// <returns>
413         /// <para>The bool</para>
414         /// <para></para>
415         /// </returns>
416         [MethodImpl(MethodImplOptions.AggressiveInlining)]
417         protected override bool FirstIsToTheLeftOfSecond(TLinkAddress first, TLinkAddress second)
418         {
419             ref var firstLink = ref LinksDataParts[first];
420             ref var secondLink = ref LinksDataParts[second];
421             return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
                ↪ secondLink.Source, secondLink.Target);
422         }
423
424         /// <summary>
425         /// <para>
426         /// Determines whether this instance first is to the right of second.
427         /// </para>
428         /// <para></para>
429         /// </summary>
430         /// <param name="first">
431         /// <para>The first.</para>
432         /// <para></para>
433         /// </param>
434         /// <param name="second">
435         /// <para>The second.</para>
436         /// <para></para>
437         /// </param>
438         /// <returns>
439         /// <para>The bool</para>
440         /// <para></para>
441         /// </returns>
442         [MethodImpl(MethodImplOptions.AggressiveInlining)]
443         protected override bool FirstIsToTheRightOfSecond(TLinkAddress first, TLinkAddress
            ↪ second)
444         {
445             ref var firstLink = ref LinksDataParts[first];
446             ref var secondLink = ref LinksDataParts[second];
447             return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
                ↪ secondLink.Source, secondLink.Target);
448         }
449     }
450 }
```

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

```
1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.Memory.Split.Generic;
3  using TLinkAddress = System.UInt32;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Memory.Split.Specific
8  {
9      /// <summary>
10     /// <para>
11     /// Represents the int 32 external links size balanced tree methods base.
12     /// </para>
13     /// <para></para>
14     /// </summary>
15     /// <seealso cref="ExternalLinksSizeBalancedTreeMethodsBase{TLinkAddress}"/>
```

```
16      /// <seealso cref="ILinksTreeMethods{TLinkAddress}"/>
17      public unsafe abstract class UInt32ExternalLinksSizeBalancedTreeMethodsBase :
        ↪  ExternalLinksSizeBalancedTreeMethodsBase<TLinkAddress>, ILinksTreeMethods<TLinkAddress>
18      {
19          /// <summary>
20          /// <para>
21          /// The links data parts.
22          /// </para>
23          /// <para></para>
24          /// </summary>
25          protected new readonly RawLinkDataPart<TLinkAddress>* LinksDataParts;
26          /// <summary>
27          /// <para>
28          /// The links index parts.
29          /// </para>
30          /// <para></para>
31          /// </summary>
32          protected new readonly RawLinkIndexPart<TLinkAddress>* LinksIndexParts;
33          /// <summary>
34          /// <para>
35          /// The header.
36          /// </para>
37          /// <para></para>
38          /// </summary>
39          protected new readonly LinksHeader<TLinkAddress>* Header;
40
41          /// <summary>
42          /// <para>
43          /// Initializes a new <see cref="UInt32ExternalLinksSizeBalancedTreeMethodsBase"/>
            ↪  instance.
44          /// </para>
45          /// <para></para>
46          /// </summary>
47          /// <param name="constants">
48          /// <para>A constants.</para>
49          /// <para></para>
50          /// </param>
51          /// <param name="linksDataParts">
52          /// <para>A links data parts.</para>
53          /// <para></para>
54          /// </param>
55          /// <param name="linksIndexParts">
56          /// <para>A links index parts.</para>
57          /// <para></para>
58          /// </param>
59          /// <param name="header">
60          /// <para>A header.</para>
61          /// <para></para>
62          /// </param>
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          protected UInt32ExternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLinkAddress>
            ↪  constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
            ↪  RawLinkIndexPart<TLinkAddress>* linksIndexParts, LinksHeader<TLinkAddress>* header)
65              : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
66          {
67              LinksDataParts = linksDataParts;
68              LinksIndexParts = linksIndexParts;
69              Header = header;
70          }
71
72          /// <summary>
73          /// <para>
74          /// Gets the zero.
75          /// </para>
76          /// <para></para>
77          /// </summary>
78          /// <returns>
79          /// <para>The link</para>
80          /// <para></para>
81          /// </returns>
82          [MethodImpl(MethodImplOptions.AggressiveInlining)]
83          protected override TLinkAddress GetZero() => 0U;
84
85          /// <summary>
86          /// <para>
87          /// Determines whether this instance equal to zero.
88          /// </para>
89          /// <para></para>
```

```
90          /// </summary>
91          /// <param name="value">
92          /// <para>The value.</para>
93          /// <para></para>
94          /// </param>
95          /// <returns>
96          /// <para>The bool</para>
97          /// <para></para>
98          /// </returns>
99          [MethodImpl(MethodImplOptions.AggressiveInlining)]
100         protected override bool EqualToZero(TLinkAddress value) => value == 0U;
101
102         /// <summary>
103         /// <para>
104         /// Determines whether this instance are equal.
105         /// </para>
106         /// <para></para>
107         /// </summary>
108         /// <param name="first">
109         /// <para>The first.</para>
110         /// <para></para>
111         /// </param>
112         /// <param name="second">
113         /// <para>The second.</para>
114         /// <para></para>
115         /// </param>
116         /// <returns>
117         /// <para>The bool</para>
118         /// <para></para>
119         /// </returns>
120         [MethodImpl(MethodImplOptions.AggressiveInlining)]
121         protected override bool AreEqual(TLinkAddress first, TLinkAddress second) => first ==
            ↪ second;
122
123         /// <summary>
124         /// <para>
125         /// Determines whether this instance greater than zero.
126         /// </para>
127         /// <para></para>
128         /// </summary>
129         /// <param name="value">
130         /// <para>The value.</para>
131         /// <para></para>
132         /// </param>
133         /// <returns>
134         /// <para>The bool</para>
135         /// <para></para>
136         /// </returns>
137         [MethodImpl(MethodImplOptions.AggressiveInlining)]
138         protected override bool GreaterThanZero(TLinkAddress value) => value > 0U;
139
140         /// <summary>
141         /// <para>
142         /// Determines whether this instance greater than.
143         /// </para>
144         /// <para></para>
145         /// </summary>
146         /// <param name="first">
147         /// <para>The first.</para>
148         /// <para></para>
149         /// </param>
150         /// <param name="second">
151         /// <para>The second.</para>
152         /// <para></para>
153         /// </param>
154         /// <returns>
155         /// <para>The bool</para>
156         /// <para></para>
157         /// </returns>
158         [MethodImpl(MethodImplOptions.AggressiveInlining)]
159         protected override bool GreaterThan(TLinkAddress first, TLinkAddress second) => first >
            ↪ second;
160
161         /// <summary>
162         /// <para>
163         /// Determines whether this instance greater or equal than.
164         /// </para>
165         /// <para></para>
```

```
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterOrEqualThan(TLinkAddress first, TLinkAddress second) =>
        ↪  first >= second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater or equal than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterOrEqualThanZero(TLinkAddress value) => true; // value >=
        ↪  0 is always true for ulong

        /// <summary>
        /// <para>
        /// Determines whether this instance less or equal than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessOrEqualThanZero(TLinkAddress value) => value == OUL; //
        ↪  value is always >= 0 for ulong

        /// <summary>
        /// <para>
        /// Determines whether this instance less or equal than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessOrEqualThan(TLinkAddress first, TLinkAddress second) =>
        ↪  first <= second;

        /// <summary>
        /// <para>
        /// Determines whether this instance less than zero.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessThanZero(TLinkAddress value) => false; // value < 0 is
        ↪  always false for ulong

        /// <summary>
        /// <para>
        /// Determines whether this instance less than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessThan(TLinkAddress first, TLinkAddress second) => first <
        ↪  second;

        /// <summary>
        /// <para>
        /// Increments the value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress Increment(TLinkAddress value) => ++value;

        /// <summary>
        /// <para>
        /// Decrements the value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress Decrement(TLinkAddress value) => --value;

        /// <summary>
        /// <para>
        /// Adds the first.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
```

```
316             /// <para>The first.</para>
317             /// <para></para>
318             /// </param>
319             /// <param name="second">
320             /// <para>The second.</para>
321             /// <para></para>
322             /// </param>
323             /// <returns>
324             /// <para>The link</para>
325             /// <para></para>
326             /// </returns>
327             [MethodImpl(MethodImplOptions.AggressiveInlining)]
328             protected override TLinkAddress Add(TLinkAddress first, TLinkAddress second) => first +
                ↪  second;
329
330             /// <summary>
331             /// <para>
332             /// Subtracts the first.
333             /// </para>
334             /// <para></para>
335             /// </summary>
336             /// <param name="first">
337             /// <para>The first.</para>
338             /// <para></para>
339             /// </param>
340             /// <param name="second">
341             /// <para>The second.</para>
342             /// <para></para>
343             /// </param>
344             /// <returns>
345             /// <para>The link</para>
346             /// <para></para>
347             /// </returns>
348             [MethodImpl(MethodImplOptions.AggressiveInlining)]
349             protected override TLinkAddress Subtract(TLinkAddress first, TLinkAddress second) =>
                ↪  first - second;
350
351             /// <summary>
352             /// <para>
353             /// Gets the header reference.
354             /// </para>
355             /// <para></para>
356             /// </summary>
357             /// <returns>
358             /// <para>A ref links header of t link</para>
359             /// <para></para>
360             /// </returns>
361             [MethodImpl(MethodImplOptions.AggressiveInlining)]
362             protected override ref LinksHeader<TLinkAddress> GetHeaderReference() => ref *Header;
363
364             /// <summary>
365             /// <para>
366             /// Gets the link data part reference using the specified link.
367             /// </para>
368             /// <para></para>
369             /// </summary>
370             /// <param name="link">
371             /// <para>The link.</para>
372             /// <para></para>
373             /// </param>
374             /// <returns>
375             /// <para>A ref raw link data part of t link</para>
376             /// <para></para>
377             /// </returns>
378             [MethodImpl(MethodImplOptions.AggressiveInlining)]
379             protected override ref RawLinkDataPart<TLinkAddress>
                ↪  GetLinkDataPartReference(TLinkAddress link) => ref LinksDataParts[link];
380
381             /// <summary>
382             /// <para>
383             /// Gets the link index part reference using the specified link.
384             /// </para>
385             /// <para></para>
386             /// </summary>
387             /// <param name="link">
388             /// <para>The link.</para>
389             /// <para></para>
390             /// </param>
```

```
        /// <returns>
        /// <para>A ref raw link index part of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkIndexPart<TLinkAddress>
        ↪  GetLinkIndexPartReference(TLinkAddress link) => ref LinksIndexParts[link];

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLinkAddress first, TLinkAddress second)
        {
            ref var firstLink = ref LinksDataParts[first];
            ref var secondLink = ref LinksDataParts[second];
            return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
            ↪  secondLink.Source, secondLink.Target);
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the right of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLinkAddress first, TLinkAddress
        ↪  second)
        {
            ref var firstLink = ref LinksDataParts[first];
            ref var secondLink = ref LinksDataParts[second];
            return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
            ↪  secondLink.Source, secondLink.Target);
        }
    }
}
```

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

```
using System.Runtime.CompilerServices;
using TLinkAddress = System.UInt32;

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    /// <summary>
    /// <para>
    /// Represents the int 32 external links sources recursionless size balanced tree methods.
    /// </para>
    /// <para></para>
```

```
13      /// </summary>
14      /// <seealso cref="UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase"/>
15      public unsafe class UInt32ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods :
        ↪  UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase
16      {
17          /// <summary>
18          /// <para>
19          /// Initializes a new <see
            ↪  cref="UInt32ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods"/> instance.
20          /// </para>
21          /// <para></para>
22          /// </summary>
23          /// <param name="constants">
24          /// <para>A constants.</para>
25          /// <para></para>
26          /// </param>
27          /// <param name="linksDataParts">
28          /// <para>A links data parts.</para>
29          /// <para></para>
30          /// </param>
31          /// <param name="linksIndexParts">
32          /// <para>A links index parts.</para>
33          /// <para></para>
34          /// </param>
35          /// <param name="header">
36          /// <para>A header.</para>
37          /// <para></para>
38          /// </param>
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          public UInt32ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<TLi
            ↪  nkAddress> constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
            ↪  RawLinkIndexPart<TLinkAddress>* linksIndexParts, LinksHeader<TLinkAddress>* header)
            ↪  : base(constants, linksDataParts, linksIndexParts, header) { }
41
42          /// <summary>
43          /// <para>
44          /// Gets the left reference using the specified node.
45          /// </para>
46          /// <para></para>
47          /// </summary>
48          /// <param name="node">
49          /// <para>The node.</para>
50          /// <para></para>
51          /// </param>
52          /// <returns>
53          /// <para>The ref link</para>
54          /// <para></para>
55          /// </returns>
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
            ↪  LinksIndexParts[node].LeftAsSource;
58
59          /// <summary>
60          /// <para>
61          /// Gets the right reference using the specified node.
62          /// </para>
63          /// <para></para>
64          /// </summary>
65          /// <param name="node">
66          /// <para>The node.</para>
67          /// <para></para>
68          /// </param>
69          /// <returns>
70          /// <para>The ref link</para>
71          /// <para></para>
72          /// </returns>
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
            ↪  LinksIndexParts[node].RightAsSource;
75
76          /// <summary>
77          /// <para>
78          /// Gets the left using the specified node.
79          /// </para>
80          /// <para></para>
81          /// </summary>
82          /// <param name="node">
83          /// <para>The node.</para>
```

```
84          /// <para></para>
85          /// </param>
86          /// <returns>
87          /// <para>The link</para>
88          /// <para></para>
89          /// </returns>
90          [MethodImpl(MethodImplOptions.AggressiveInlining)]
91          protected override TLinkAddress GetLeft(TLinkAddress node) =>
            ↪ LinksIndexParts[node].LeftAsSource;
92
93          /// <summary>
94          /// <para>
95          /// Gets the right using the specified node.
96          /// </para>
97          /// <para></para>
98          /// </summary>
99          /// <param name="node">
100         /// <para>The node.</para>
101         /// <para></para>
102         /// </param>
103         /// <returns>
104         /// <para>The link</para>
105         /// <para></para>
106         /// </returns>
107         [MethodImpl(MethodImplOptions.AggressiveInlining)]
108         protected override TLinkAddress GetRight(TLinkAddress node) =>
            ↪ LinksIndexParts[node].RightAsSource;
109
110         /// <summary>
111         /// <para>
112         /// Sets the left using the specified node.
113         /// </para>
114         /// <para></para>
115         /// </summary>
116         /// <param name="node">
117         /// <para>The node.</para>
118         /// <para></para>
119         /// </param>
120         /// <param name="left">
121         /// <para>The left.</para>
122         /// <para></para>
123         /// </param>
124         [MethodImpl(MethodImplOptions.AggressiveInlining)]
125         protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
            ↪ LinksIndexParts[node].LeftAsSource = left;
126
127         /// <summary>
128         /// <para>
129         /// Sets the right using the specified node.
130         /// </para>
131         /// <para></para>
132         /// </summary>
133         /// <param name="node">
134         /// <para>The node.</para>
135         /// <para></para>
136         /// </param>
137         /// <param name="right">
138         /// <para>The right.</para>
139         /// <para></para>
140         /// </param>
141         [MethodImpl(MethodImplOptions.AggressiveInlining)]
142         protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
            ↪ LinksIndexParts[node].RightAsSource = right;
143
144         /// <summary>
145         /// <para>
146         /// Gets the size using the specified node.
147         /// </para>
148         /// <para></para>
149         /// </summary>
150         /// <param name="node">
151         /// <para>The node.</para>
152         /// <para></para>
153         /// </param>
154         /// <returns>
155         /// <para>The link</para>
156         /// <para></para>
157         /// </returns>
```

```
158         [MethodImpl(MethodImplOptions.AggressiveInlining)]
159         protected override TLinkAddress GetSize(TLinkAddress node) =>
            ↪ LinksIndexParts[node].SizeAsSource;
160
161         /// <summary>
162         /// <para>
163         /// Sets the size using the specified node.
164         /// </para>
165         /// <para></para>
166         /// </summary>
167         /// <param name="node">
168         /// <para>The node.</para>
169         /// <para></para>
170         /// </param>
171         /// <param name="size">
172         /// <para>The size.</para>
173         /// <para></para>
174         /// </param>
175         [MethodImpl(MethodImplOptions.AggressiveInlining)]
176         protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
            ↪ LinksIndexParts[node].SizeAsSource = size;
177
178         /// <summary>
179         /// <para>
180         /// Gets the tree root.
181         /// </para>
182         /// <para></para>
183         /// </summary>
184         /// <returns>
185         /// <para>The link</para>
186         /// <para></para>
187         /// </returns>
188         [MethodImpl(MethodImplOptions.AggressiveInlining)]
189         protected override TLinkAddress GetTreeRoot() => Header->RootAsSource;
190
191         /// <summary>
192         /// <para>
193         /// Gets the base part value using the specified node.
194         /// </para>
195         /// <para></para>
196         /// </summary>
197         /// <param name="node">
198         /// <para>The node.</para>
199         /// <para></para>
200         /// </param>
201         /// <returns>
202         /// <para>The link</para>
203         /// <para></para>
204         /// </returns>
205         [MethodImpl(MethodImplOptions.AggressiveInlining)]
206         protected override TLinkAddress GetBasePartValue(TLinkAddress node) =>
            ↪ LinksDataParts[node].Source;
207
208         /// <summary>
209         /// <para>
210         /// Determines whether this instance first is to the left of second.
211         /// </para>
212         /// <para></para>
213         /// </summary>
214         /// <param name="firstSource">
215         /// <para>The first source.</para>
216         /// <para></para>
217         /// </param>
218         /// <param name="firstTarget">
219         /// <para>The first target.</para>
220         /// <para></para>
221         /// </param>
222         /// <param name="secondSource">
223         /// <para>The second source.</para>
224         /// <para></para>
225         /// </param>
226         /// <param name="secondTarget">
227         /// <para>The second target.</para>
228         /// <para></para>
229         /// </param>
230         /// <returns>
231         /// <para>The bool</para>
232         /// <para></para>
```

```
233             /// </returns>
234             [MethodImpl(MethodImplOptions.AggressiveInlining)]
235             protected override bool FirstIsToTheLeftOfSecond(TLinkAddress firstSource, TLinkAddress
                ↪  firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget)
236                 => firstSource < secondSource || firstSource == secondSource && firstTarget <
                    ↪  secondTarget;
237
238             /// <summary>
239             /// <para>
240             /// Determines whether this instance first is to the right of second.
241             /// </para>
242             /// <para></para>
243             /// </summary>
244             /// <param name="firstSource">
245             /// <para>The first source.</para>
246             /// <para></para>
247             /// </param>
248             /// <param name="firstTarget">
249             /// <para>The first target.</para>
250             /// <para></para>
251             /// </param>
252             /// <param name="secondSource">
253             /// <para>The second source.</para>
254             /// <para></para>
255             /// </param>
256             /// <param name="secondTarget">
257             /// <para>The second target.</para>
258             /// <para></para>
259             /// </param>
260             /// <returns>
261             /// <para>The bool</para>
262             /// <para></para>
263             /// </returns>
264             [MethodImpl(MethodImplOptions.AggressiveInlining)]
265             protected override bool FirstIsToTheRightOfSecond(TLinkAddress firstSource, TLinkAddress
                ↪  firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget)
266                 => firstSource > secondSource || firstSource == secondSource && firstTarget >
                    ↪  secondTarget;
267
268             /// <summary>
269             /// <para>
270             /// Clears the node using the specified node.
271             /// </para>
272             /// <para></para>
273             /// </summary>
274             /// <param name="node">
275             /// <para>The node.</para>
276             /// <para></para>
277             /// </param>
278             [MethodImpl(MethodImplOptions.AggressiveInlining)]
279             protected override void ClearNode(TLinkAddress node)
280             {
281                 ref var link = ref LinksIndexParts[node];
282                 link.LeftAsSource = Zero;
283                 link.RightAsSource = Zero;
284                 link.SizeAsSource = Zero;
285             }
286         }
287     }
```

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

```
1   using System.Runtime.CompilerServices;
2   using TLinkAddress = System.UInt32;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Memory.Split.Specific
7   {
8       /// <summary>
9       /// <para>
10      /// Represents the int 32 external links sources size balanced tree methods.
11      /// </para>
12      /// <para></para>
13      /// </summary>
14      /// <seealso cref="UInt32ExternalLinksSizeBalancedTreeMethodsBase"/>
15      public unsafe class UInt32ExternalLinksSourcesSizeBalancedTreeMethods :
            ↪  UInt32ExternalLinksSizeBalancedTreeMethodsBase
16      {
```

```
17          /// <summary>
18          /// <para>
19          /// Initializes a new <see cref="UInt32ExternalLinksSourcesSizeBalancedTreeMethods"/>
            ↪  instance.
20          /// </para>
21          /// <para></para>
22          /// </summary>
23          /// <param name="constants">
24          /// <para>A constants.</para>
25          /// <para></para>
26          /// </param>
27          /// <param name="linksDataParts">
28          /// <para>A links data parts.</para>
29          /// <para></para>
30          /// </param>
31          /// <param name="linksIndexParts">
32          /// <para>A links index parts.</para>
33          /// <para></para>
34          /// </param>
35          /// <param name="header">
36          /// <para>A header.</para>
37          /// <para></para>
38          /// </param>
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          public UInt32ExternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLinkAddress>
            ↪  constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
            ↪  RawLinkIndexPart<TLinkAddress>* linksIndexParts, LinksHeader<TLinkAddress>* header)
            ↪  : base(constants, linksDataParts, linksIndexParts, header) { }
41
42          /// <summary>
43          /// <para>
44          /// Gets the left reference using the specified node.
45          /// </para>
46          /// <para></para>
47          /// </summary>
48          /// <param name="node">
49          /// <para>The node.</para>
50          /// <para></para>
51          /// </param>
52          /// <returns>
53          /// <para>The ref link</para>
54          /// <para></para>
55          /// </returns>
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
            ↪  LinksIndexParts[node].LeftAsSource;
58
59          /// <summary>
60          /// <para>
61          /// Gets the right reference using the specified node.
62          /// </para>
63          /// <para></para>
64          /// </summary>
65          /// <param name="node">
66          /// <para>The node.</para>
67          /// <para></para>
68          /// </param>
69          /// <returns>
70          /// <para>The ref link</para>
71          /// <para></para>
72          /// </returns>
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
            ↪  LinksIndexParts[node].RightAsSource;
75
76          /// <summary>
77          /// <para>
78          /// Gets the left using the specified node.
79          /// </para>
80          /// <para></para>
81          /// </summary>
82          /// <param name="node">
83          /// <para>The node.</para>
84          /// <para></para>
85          /// </param>
86          /// <returns>
87          /// <para>The link</para>
88          /// <para></para>
```

```
89          /// </returns>
90          [MethodImpl(MethodImplOptions.AggressiveInlining)]
91          protected override TLinkAddress GetLeft(TLinkAddress node) =>
            ↪ LinksIndexParts[node].LeftAsSource;
92
93          /// <summary>
94          /// <para>
95          /// Gets the right using the specified node.
96          /// </para>
97          /// <para></para>
98          /// </summary>
99          /// <param name="node">
100         /// <para>The node.</para>
101         /// <para></para>
102         /// </param>
103         /// <returns>
104         /// <para>The link</para>
105         /// <para></para>
106         /// </returns>
107         [MethodImpl(MethodImplOptions.AggressiveInlining)]
108         protected override TLinkAddress GetRight(TLinkAddress node) =>
            ↪ LinksIndexParts[node].RightAsSource;
109
110         /// <summary>
111         /// <para>
112         /// Sets the left using the specified node.
113         /// </para>
114         /// <para></para>
115         /// </summary>
116         /// <param name="node">
117         /// <para>The node.</para>
118         /// <para></para>
119         /// </param>
120         /// <param name="left">
121         /// <para>The left.</para>
122         /// <para></para>
123         /// </param>
124         [MethodImpl(MethodImplOptions.AggressiveInlining)]
125         protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
            ↪ LinksIndexParts[node].LeftAsSource = left;
126
127         /// <summary>
128         /// <para>
129         /// Sets the right using the specified node.
130         /// </para>
131         /// <para></para>
132         /// </summary>
133         /// <param name="node">
134         /// <para>The node.</para>
135         /// <para></para>
136         /// </param>
137         /// <param name="right">
138         /// <para>The right.</para>
139         /// <para></para>
140         /// </param>
141         [MethodImpl(MethodImplOptions.AggressiveInlining)]
142         protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
            ↪ LinksIndexParts[node].RightAsSource = right;
143
144         /// <summary>
145         /// <para>
146         /// Gets the size using the specified node.
147         /// </para>
148         /// <para></para>
149         /// </summary>
150         /// <param name="node">
151         /// <para>The node.</para>
152         /// <para></para>
153         /// </param>
154         /// <returns>
155         /// <para>The link</para>
156         /// <para></para>
157         /// </returns>
158         [MethodImpl(MethodImplOptions.AggressiveInlining)]
159         protected override TLinkAddress GetSize(TLinkAddress node) =>
            ↪ LinksIndexParts[node].SizeAsSource;
160
```

```
        /// <summary>
        /// <para>
        /// Sets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="size">
        /// <para>The size.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
        ↪ LinksIndexParts[node].SizeAsSource = size;

        /// <summary>
        /// <para>
        /// Gets the tree root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetTreeRoot() => Header->RootAsSource;

        /// <summary>
        /// <para>
        /// Gets the base part value using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetBasePartValue(TLinkAddress node) =>
        ↪ LinksDataParts[node].Source;

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="firstSource">
        /// <para>The first source.</para>
        /// <para></para>
        /// </param>
        /// <param name="firstTarget">
        /// <para>The first target.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondSource">
        /// <para>The second source.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondTarget">
        /// <para>The second target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLinkAddress firstSource, TLinkAddress
        ↪ firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget)
```

```
            => firstSource < secondSource || firstSource == secondSource && firstTarget <
            ↪  secondTarget;

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the right of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="firstSource">
        /// <para>The first source.</para>
        /// <para></para>
        /// </param>
        /// <param name="firstTarget">
        /// <para>The first target.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondSource">
        /// <para>The second source.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondTarget">
        /// <para>The second target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLinkAddress firstSource, TLinkAddress
        ↪  firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget)
            => firstSource > secondSource || firstSource == secondSource && firstTarget >
            ↪  secondTarget;

        /// <summary>
        /// <para>
        /// Clears the node using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLinkAddress node)
        {
            ref var link = ref LinksIndexParts[node];
            link.LeftAsSource = Zero;
            link.RightAsSource = Zero;
            link.SizeAsSource = Zero;
        }
    }
}
```

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

```
using System.Runtime.CompilerServices;
using TLinkAddress = System.UInt32;

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    /// <summary>
    /// <para>
    /// Represents the int 32 external links targets recursionless size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase"/>
    public unsafe class UInt32ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods :
    ↪  UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see
        ↪  cref="UInt32ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods"/> instance.
```

```
20          /// </para>
21          /// <para></para>
22          /// </summary>
23          /// <param name="constants">
24          /// <para>A constants.</para>
25          /// <para></para>
26          /// </param>
27          /// <param name="linksDataParts">
28          /// <para>A links data parts.</para>
29          /// <para></para>
30          /// </param>
31          /// <param name="linksIndexParts">
32          /// <para>A links index parts.</para>
33          /// <para></para>
34          /// </param>
35          /// <param name="header">
36          /// <para>A header.</para>
37          /// <para></para>
38          /// </param>
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          public UInt32ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<TLi
            ↪  nkAddress> constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
            ↪  RawLinkIndexPart<TLinkAddress>* linksIndexParts, LinksHeader<TLinkAddress>* header)
            ↪  : base(constants, linksDataParts, linksIndexParts, header) { }
41
42          /// <summary>
43          /// <para>
44          /// Gets the left reference using the specified node.
45          /// </para>
46          /// <para></para>
47          /// </summary>
48          /// <param name="node">
49          /// <para>The node.</para>
50          /// <para></para>
51          /// </param>
52          /// <returns>
53          /// <para>The ref link</para>
54          /// <para></para>
55          /// </returns>
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
            ↪  LinksIndexParts[node].LeftAsTarget;
58
59          /// <summary>
60          /// <para>
61          /// Gets the right reference using the specified node.
62          /// </para>
63          /// <para></para>
64          /// </summary>
65          /// <param name="node">
66          /// <para>The node.</para>
67          /// <para></para>
68          /// </param>
69          /// <returns>
70          /// <para>The ref link</para>
71          /// <para></para>
72          /// </returns>
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
            ↪  LinksIndexParts[node].RightAsTarget;
75
76          /// <summary>
77          /// <para>
78          /// Gets the left using the specified node.
79          /// </para>
80          /// <para></para>
81          /// </summary>
82          /// <param name="node">
83          /// <para>The node.</para>
84          /// <para></para>
85          /// </param>
86          /// <returns>
87          /// <para>The link</para>
88          /// <para></para>
89          /// </returns>
90          [MethodImpl(MethodImplOptions.AggressiveInlining)]
91          protected override TLinkAddress GetLeft(TLinkAddress node) =>
            ↪  LinksIndexParts[node].LeftAsTarget;
```

```
92
93          /// <summary>
94          /// <para>
95          /// Gets the right using the specified node.
96          /// </para>
97          /// <para></para>
98          /// </summary>
99          /// <param name="node">
100         /// <para>The node.</para>
101         /// <para></para>
102         /// </param>
103         /// <returns>
104         /// <para>The link</para>
105         /// <para></para>
106         /// </returns>
107         [MethodImpl(MethodImplOptions.AggressiveInlining)]
108         protected override TLinkAddress GetRight(TLinkAddress node) =>
            ↪ LinksIndexParts[node].RightAsTarget;
109
110         /// <summary>
111         /// <para>
112         /// Sets the left using the specified node.
113         /// </para>
114         /// <para></para>
115         /// </summary>
116         /// <param name="node">
117         /// <para>The node.</para>
118         /// <para></para>
119         /// </param>
120         /// <param name="left">
121         /// <para>The left.</para>
122         /// <para></para>
123         /// </param>
124         [MethodImpl(MethodImplOptions.AggressiveInlining)]
125         protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
            ↪ LinksIndexParts[node].LeftAsTarget = left;
126
127         /// <summary>
128         /// <para>
129         /// Sets the right using the specified node.
130         /// </para>
131         /// <para></para>
132         /// </summary>
133         /// <param name="node">
134         /// <para>The node.</para>
135         /// <para></para>
136         /// </param>
137         /// <param name="right">
138         /// <para>The right.</para>
139         /// <para></para>
140         /// </param>
141         [MethodImpl(MethodImplOptions.AggressiveInlining)]
142         protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
            ↪ LinksIndexParts[node].RightAsTarget = right;
143
144         /// <summary>
145         /// <para>
146         /// Gets the size using the specified node.
147         /// </para>
148         /// <para></para>
149         /// </summary>
150         /// <param name="node">
151         /// <para>The node.</para>
152         /// <para></para>
153         /// </param>
154         /// <returns>
155         /// <para>The link</para>
156         /// <para></para>
157         /// </returns>
158         [MethodImpl(MethodImplOptions.AggressiveInlining)]
159         protected override TLinkAddress GetSize(TLinkAddress node) =>
            ↪ LinksIndexParts[node].SizeAsTarget;
160
161         /// <summary>
162         /// <para>
163         /// Sets the size using the specified node.
164         /// </para>
165         /// <para></para>
```

```
166         /// </summary>
167         /// <param name="node">
168         /// <para>The node.</para>
169         /// <para></para>
170         /// </param>
171         /// <param name="size">
172         /// <para>The size.</para>
173         /// <para></para>
174         /// </param>
175         [MethodImpl(MethodImplOptions.AggressiveInlining)]
176         protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
            ↪ LinksIndexParts[node].SizeAsTarget = size;
177
178         /// <summary>
179         /// <para>
180         /// Gets the tree root.
181         /// </para>
182         /// <para></para>
183         /// </summary>
184         /// <returns>
185         /// <para>The link</para>
186         /// <para></para>
187         /// </returns>
188         [MethodImpl(MethodImplOptions.AggressiveInlining)]
189         protected override TLinkAddress GetTreeRoot() => Header->RootAsTarget;
190
191         /// <summary>
192         /// <para>
193         /// Gets the base part value using the specified node.
194         /// </para>
195         /// <para></para>
196         /// </summary>
197         /// <param name="node">
198         /// <para>The node.</para>
199         /// <para></para>
200         /// </param>
201         /// <returns>
202         /// <para>The link</para>
203         /// <para></para>
204         /// </returns>
205         [MethodImpl(MethodImplOptions.AggressiveInlining)]
206         protected override TLinkAddress GetBasePartValue(TLinkAddress node) =>
            ↪ LinksDataParts[node].Target;
207
208         /// <summary>
209         /// <para>
210         /// Determines whether this instance first is to the left of second.
211         /// </para>
212         /// <para></para>
213         /// </summary>
214         /// <param name="firstSource">
215         /// <para>The first source.</para>
216         /// <para></para>
217         /// </param>
218         /// <param name="firstTarget">
219         /// <para>The first target.</para>
220         /// <para></para>
221         /// </param>
222         /// <param name="secondSource">
223         /// <para>The second source.</para>
224         /// <para></para>
225         /// </param>
226         /// <param name="secondTarget">
227         /// <para>The second target.</para>
228         /// <para></para>
229         /// </param>
230         /// <returns>
231         /// <para>The bool</para>
232         /// <para></para>
233         /// </returns>
234         [MethodImpl(MethodImplOptions.AggressiveInlining)]
235         protected override bool FirstIsToTheLeftOfSecond(TLinkAddress firstSource, TLinkAddress
            ↪ firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget)
                => firstTarget < secondTarget || firstTarget == secondTarget && firstSource <
                ↪ secondSource;
237
238         /// <summary>
239         /// <para>
```

```
240             /// Determines whether this instance first is to the right of second.
241             /// </para>
242             /// <para></para>
243             /// </summary>
244             /// <param name="firstSource">
245             /// <para>The first source.</para>
246             /// <para></para>
247             /// </param>
248             /// <param name="firstTarget">
249             /// <para>The first target.</para>
250             /// <para></para>
251             /// </param>
252             /// <param name="secondSource">
253             /// <para>The second source.</para>
254             /// <para></para>
255             /// </param>
256             /// <param name="secondTarget">
257             /// <para>The second target.</para>
258             /// <para></para>
259             /// </param>
260             /// <returns>
261             /// <para>The bool</para>
262             /// <para></para>
263             /// </returns>
264             [MethodImpl(MethodImplOptions.AggressiveInlining)]
265             protected override bool FirstIsToTheRightOfSecond(TLinkAddress firstSource, TLinkAddress
                ↪  firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget)
266                 => firstTarget > secondTarget || firstTarget == secondTarget && firstSource >
                    ↪  secondSource;
267
268             /// <summary>
269             /// <para>
270             /// Clears the node using the specified node.
271             /// </para>
272             /// <para></para>
273             /// </summary>
274             /// <param name="node">
275             /// <para>The node.</para>
276             /// <para></para>
277             /// </param>
278             [MethodImpl(MethodImplOptions.AggressiveInlining)]
279             protected override void ClearNode(TLinkAddress node)
280             {
281                 ref var link = ref LinksIndexParts[node];
282                 link.LeftAsTarget = Zero;
283                 link.RightAsTarget = Zero;
284                 link.SizeAsTarget = Zero;
285             }
286         }
287     }
```

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

```
1   using System.Runtime.CompilerServices;
2   using TLinkAddress = System.UInt32;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Memory.Split.Specific
7   {
8       /// <summary>
9       /// <para>
10      /// Represents the int 32 external links targets size balanced tree methods.
11      /// </para>
12      /// <para></para>
13      /// </summary>
14      /// <seealso cref="UInt32ExternalLinksSizeBalancedTreeMethodsBase"/>
15      public unsafe class UInt32ExternalLinksTargetsSizeBalancedTreeMethods :
            ↪  UInt32ExternalLinksSizeBalancedTreeMethodsBase
16      {
17          /// <summary>
18          /// <para>
19          /// Initializes a new <see cref="UInt32ExternalLinksTargetsSizeBalancedTreeMethods"/>
                ↪  instance.
20          /// </para>
21          /// <para></para>
22          /// </summary>
23          /// <param name="constants">
24          /// <para>A constants.</para>
```

```
25          /// <para></para>
26          /// </param>
27          /// <param name="linksDataParts">
28          /// <para>A links data parts.</para>
29          /// <para></para>
30          /// </param>
31          /// <param name="linksIndexParts">
32          /// <para>A links index parts.</para>
33          /// <para></para>
34          /// </param>
35          /// <param name="header">
36          /// <para>A header.</para>
37          /// <para></para>
38          /// </param>
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          public UInt32ExternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLinkAddress>
            ↪  constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
            ↪  RawLinkIndexPart<TLinkAddress>* linksIndexParts, LinksHeader<TLinkAddress>* header)
            ↪  : base(constants, linksDataParts, linksIndexParts, header) { }
41
42          /// <summary>
43          /// <para>
44          /// Gets the left reference using the specified node.
45          /// </para>
46          /// <para></para>
47          /// </summary>
48          /// <param name="node">
49          /// <para>The node.</para>
50          /// <para></para>
51          /// </param>
52          /// <returns>
53          /// <para>The ref link</para>
54          /// <para></para>
55          /// </returns>
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
            ↪  LinksIndexParts[node].LeftAsTarget;
58
59          /// <summary>
60          /// <para>
61          /// Gets the right reference using the specified node.
62          /// </para>
63          /// <para></para>
64          /// </summary>
65          /// <param name="node">
66          /// <para>The node.</para>
67          /// <para></para>
68          /// </param>
69          /// <returns>
70          /// <para>The ref link</para>
71          /// <para></para>
72          /// </returns>
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
            ↪  LinksIndexParts[node].RightAsTarget;
75
76          /// <summary>
77          /// <para>
78          /// Gets the left using the specified node.
79          /// </para>
80          /// <para></para>
81          /// </summary>
82          /// <param name="node">
83          /// <para>The node.</para>
84          /// <para></para>
85          /// </param>
86          /// <returns>
87          /// <para>The link</para>
88          /// <para></para>
89          /// </returns>
90          [MethodImpl(MethodImplOptions.AggressiveInlining)]
91          protected override TLinkAddress GetLeft(TLinkAddress node) =>
            ↪  LinksIndexParts[node].LeftAsTarget;
92
93          /// <summary>
94          /// <para>
95          /// Gets the right using the specified node.
96          /// </para>
```

```
97          /// <para></para>
98          /// </summary>
99          /// <param name="node">
100         /// <para>The node.</para>
101         /// <para></para>
102         /// </param>
103         /// <returns>
104         /// <para>The link</para>
105         /// <para></para>
106         /// </returns>
107         [MethodImpl(MethodImplOptions.AggressiveInlining)]
108         protected override TLinkAddress GetRight(TLinkAddress node) =>
            ↪ LinksIndexParts[node].RightAsTarget;
109
110         /// <summary>
111         /// <para>
112         /// Sets the left using the specified node.
113         /// </para>
114         /// <para></para>
115         /// </summary>
116         /// <param name="node">
117         /// <para>The node.</para>
118         /// <para></para>
119         /// </param>
120         /// <param name="left">
121         /// <para>The left.</para>
122         /// <para></para>
123         /// </param>
124         [MethodImpl(MethodImplOptions.AggressiveInlining)]
125         protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
            ↪ LinksIndexParts[node].LeftAsTarget = left;
126
127         /// <summary>
128         /// <para>
129         /// Sets the right using the specified node.
130         /// </para>
131         /// <para></para>
132         /// </summary>
133         /// <param name="node">
134         /// <para>The node.</para>
135         /// <para></para>
136         /// </param>
137         /// <param name="right">
138         /// <para>The right.</para>
139         /// <para></para>
140         /// </param>
141         [MethodImpl(MethodImplOptions.AggressiveInlining)]
142         protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
            ↪ LinksIndexParts[node].RightAsTarget = right;
143
144         /// <summary>
145         /// <para>
146         /// Gets the size using the specified node.
147         /// </para>
148         /// <para></para>
149         /// </summary>
150         /// <param name="node">
151         /// <para>The node.</para>
152         /// <para></para>
153         /// </param>
154         /// <returns>
155         /// <para>The link</para>
156         /// <para></para>
157         /// </returns>
158         [MethodImpl(MethodImplOptions.AggressiveInlining)]
159         protected override TLinkAddress GetSize(TLinkAddress node) =>
            ↪ LinksIndexParts[node].SizeAsTarget;
160
161         /// <summary>
162         /// <para>
163         /// Sets the size using the specified node.
164         /// </para>
165         /// <para></para>
166         /// </summary>
167         /// <param name="node">
168         /// <para>The node.</para>
169         /// <para></para>
170         /// </param>
```

```
171         /// <param name="size">
172         /// <para>The size.</para>
173         /// <para></para>
174         /// </param>
175         [MethodImpl(MethodImplOptions.AggressiveInlining)]
176         protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
            ↪ LinksIndexParts[node].SizeAsTarget = size;
177
178         /// <summary>
179         /// <para>
180         /// Gets the tree root.
181         /// </para>
182         /// <para></para>
183         /// </summary>
184         /// <returns>
185         /// <para>The link</para>
186         /// <para></para>
187         /// </returns>
188         [MethodImpl(MethodImplOptions.AggressiveInlining)]
189         protected override TLinkAddress GetTreeRoot() => Header->RootAsTarget;
190
191         /// <summary>
192         /// <para>
193         /// Gets the base part value using the specified node.
194         /// </para>
195         /// <para></para>
196         /// </summary>
197         /// <param name="node">
198         /// <para>The node.</para>
199         /// <para></para>
200         /// </param>
201         /// <returns>
202         /// <para>The link</para>
203         /// <para></para>
204         /// </returns>
205         [MethodImpl(MethodImplOptions.AggressiveInlining)]
206         protected override TLinkAddress GetBasePartValue(TLinkAddress node) =>
            ↪ LinksDataParts[node].Target;
207
208         /// <summary>
209         /// <para>
210         /// Determines whether this instance first is to the left of second.
211         /// </para>
212         /// <para></para>
213         /// </summary>
214         /// <param name="firstSource">
215         /// <para>The first source.</para>
216         /// <para></para>
217         /// </param>
218         /// <param name="firstTarget">
219         /// <para>The first target.</para>
220         /// <para></para>
221         /// </param>
222         /// <param name="secondSource">
223         /// <para>The second source.</para>
224         /// <para></para>
225         /// </param>
226         /// <param name="secondTarget">
227         /// <para>The second target.</para>
228         /// <para></para>
229         /// </param>
230         /// <returns>
231         /// <para>The bool</para>
232         /// <para></para>
233         /// </returns>
234         [MethodImpl(MethodImplOptions.AggressiveInlining)]
235         protected override bool FirstIsToTheLeftOfSecond(TLinkAddress firstSource, TLinkAddress
            ↪ firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget)
                => firstTarget < secondTarget || firstTarget == secondTarget && firstSource <
                ↪ secondSource;
237
238         /// <summary>
239         /// <para>
240         /// Determines whether this instance first is to the right of second.
241         /// </para>
242         /// <para></para>
243         /// </summary>
244         /// <param name="firstSource">
```

```
        /// <para>The first source.</para>
        /// <para></para>
        /// </param>
        /// <param name="firstTarget">
        /// <para>The first target.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondSource">
        /// <para>The second source.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondTarget">
        /// <para>The second target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLinkAddress firstSource, TLinkAddress
        ↪  firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget)
            => firstTarget > secondTarget || firstTarget == secondTarget && firstSource >
            ↪  secondSource;

        /// <summary>
        /// <para>
        /// Clears the node using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLinkAddress node)
        {
            ref var link = ref LinksIndexParts[node];
            link.LeftAsTarget = Zero;
            link.RightAsTarget = Zero;
            link.SizeAsTarget = Zero;
        }
    }
}
```

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

```
using System.Runtime.CompilerServices;
using Platform.Data.Doublets.Memory.Split.Generic;
using TLinkAddress = System.UInt32;

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    /// <summary>
    /// <para>
    /// Represents the int 32 internal links recursionless size balanced tree methods base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="InternalLinksRecursionlessSizeBalancedTreeMethodsBase{TLinkAddress}"/>
    public unsafe abstract class UInt32InternalLinksRecursionlessSizeBalancedTreeMethodsBase :
    ↪  InternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// The links data parts.
        /// </para>
        /// <para></para>
        /// </summary>
        protected new readonly RawLinkDataPart<TLinkAddress>* LinksDataParts;
        /// <summary>
        /// <para>
        /// The links index parts.
        /// </para>
        /// <para></para>
        /// </summary>
        protected new readonly RawLinkIndexPart<TLinkAddress>* LinksIndexParts;
```

```
        /// <summary>
        /// <para>
        /// The header.
        /// </para>
        /// <para></para>
        /// </summary>
        protected new readonly LinksHeader<TLinkAddress>* Header;

        /// <summary>
        /// <para>
        /// Initializes a new <see
        ↪  cref="UInt32InternalLinksRecursionlessSizeBalancedTreeMethodsBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksDataParts">
        /// <para>A links data parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksIndexParts">
        /// <para>A links index parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected UInt32InternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLi
        ↪  nkAddress> constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
        ↪  RawLinkIndexPart<TLinkAddress>* linksIndexParts, LinksHeader<TLinkAddress>* header)
            : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
        {
            LinksDataParts = linksDataParts;
            LinksIndexParts = linksIndexParts;
            Header = header;
        }

        /// <summary>
        /// <para>
        /// Gets the zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetZero() => 0U;

        /// <summary>
        /// <para>
        /// Determines whether this instance equal to zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool EqualToZero(TLinkAddress value) => value == 0U;

        /// <summary>
        /// <para>
        /// Determines whether this instance are equal.
        /// </para>
        /// <para></para>
        /// </summary>
```

```
107         /// <param name="first">
108         /// <para>The first.</para>
109         /// <para></para>
110         /// </param>
111         /// <param name="second">
112         /// <para>The second.</para>
113         /// <para></para>
114         /// </param>
115         /// <returns>
116         /// <para>The bool</para>
117         /// <para></para>
118         /// </returns>
119         [MethodImpl(MethodImplOptions.AggressiveInlining)]
120         protected override bool AreEqual(TLinkAddress first, TLinkAddress second) => first ==
            ↪ second;
121
122         /// <summary>
123         /// <para>
124         /// Determines whether this instance greater than zero.
125         /// </para>
126         /// <para></para>
127         /// </summary>
128         /// <param name="value">
129         /// <para>The value.</para>
130         /// <para></para>
131         /// </param>
132         /// <returns>
133         /// <para>The bool</para>
134         /// <para></para>
135         /// </returns>
136         [MethodImpl(MethodImplOptions.AggressiveInlining)]
137         protected override bool GreaterThanZero(TLinkAddress value) => value > 0U;
138
139         /// <summary>
140         /// <para>
141         /// Determines whether this instance greater than.
142         /// </para>
143         /// <para></para>
144         /// </summary>
145         /// <param name="first">
146         /// <para>The first.</para>
147         /// <para></para>
148         /// </param>
149         /// <param name="second">
150         /// <para>The second.</para>
151         /// <para></para>
152         /// </param>
153         /// <returns>
154         /// <para>The bool</para>
155         /// <para></para>
156         /// </returns>
157         [MethodImpl(MethodImplOptions.AggressiveInlining)]
158         protected override bool GreaterThan(TLinkAddress first, TLinkAddress second) => first >
            ↪ second;
159
160         /// <summary>
161         /// <para>
162         /// Determines whether this instance greater or equal than.
163         /// </para>
164         /// <para></para>
165         /// </summary>
166         /// <param name="first">
167         /// <para>The first.</para>
168         /// <para></para>
169         /// </param>
170         /// <param name="second">
171         /// <para>The second.</para>
172         /// <para></para>
173         /// </param>
174         /// <returns>
175         /// <para>The bool</para>
176         /// <para></para>
177         /// </returns>
178         [MethodImpl(MethodImplOptions.AggressiveInlining)]
179         protected override bool GreaterOrEqualThan(TLinkAddress first, TLinkAddress second) =>
            ↪ first >= second;
180
181         /// <summary>
```

```
        /// <para>
        /// Determines whether this instance greater or equal than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterOrEqualThanZero(TLinkAddress value) => true; // value >=
        ↪  0 is always true for ulong

        /// <summary>
        /// <para>
        /// Determines whether this instance less or equal than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessOrEqualThanZero(TLinkAddress value) => value == 0UL; //
        ↪  value is always >= 0 for ulong

        /// <summary>
        /// <para>
        /// Determines whether this instance less or equal than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessOrEqualThan(TLinkAddress first, TLinkAddress second) =>
        ↪  first <= second;

        /// <summary>
        /// <para>
        /// Determines whether this instance less than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessThanZero(TLinkAddress value) => false; // value < 0 is
        ↪  always false for ulong

        /// <summary>
        /// <para>
        /// Determines whether this instance less than.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessThan(TLinkAddress first, TLinkAddress second) => first <
        ↪  second;

        /// <summary>
        /// <para>
        /// Increments the value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress Increment(TLinkAddress value) => ++value;

        /// <summary>
        /// <para>
        /// Decrements the value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress Decrement(TLinkAddress value) => --value;

        /// <summary>
        /// <para>
        /// Adds the first.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress Add(TLinkAddress first, TLinkAddress second) => first +
        ↪  second;

        /// <summary>
        /// <para>
        /// Subtracts the first.
```

```
332         /// </para>
333         /// <para></para>
334         /// </summary>
335         /// <param name="first">
336         /// <para>The first.</para>
337         /// <para></para>
338         /// </param>
339         /// <param name="second">
340         /// <para>The second.</para>
341         /// <para></para>
342         /// </param>
343         /// <returns>
344         /// <para>The link</para>
345         /// <para></para>
346         /// </returns>
347         [MethodImpl(MethodImplOptions.AggressiveInlining)]
348         protected override TLinkAddress Subtract(TLinkAddress first, TLinkAddress second) =>
            ↪ first - second;
349
350         /// <summary>
351         /// <para>
352         /// Gets the link data part reference using the specified link.
353         /// </para>
354         /// <para></para>
355         /// </summary>
356         /// <param name="link">
357         /// <para>The link.</para>
358         /// <para></para>
359         /// </param>
360         /// <returns>
361         /// <para>A ref raw link data part of t link</para>
362         /// <para></para>
363         /// </returns>
364         [MethodImpl(MethodImplOptions.AggressiveInlining)]
365         protected override ref RawLinkDataPart<TLinkAddress>
            ↪ GetLinkDataPartReference(TLinkAddress link) => ref LinksDataParts[link];
366
367         /// <summary>
368         /// <para>
369         /// Gets the link index part reference using the specified link.
370         /// </para>
371         /// <para></para>
372         /// </summary>
373         /// <param name="link">
374         /// <para>The link.</para>
375         /// <para></para>
376         /// </param>
377         /// <returns>
378         /// <para>A ref raw link index part of t link</para>
379         /// <para></para>
380         /// </returns>
381         [MethodImpl(MethodImplOptions.AggressiveInlining)]
382         protected override ref RawLinkIndexPart<TLinkAddress>
            ↪ GetLinkIndexPartReference(TLinkAddress link) => ref LinksIndexParts[link];
383
384         /// <summary>
385         /// <para>
386         /// Determines whether this instance first is to the left of second.
387         /// </para>
388         /// <para></para>
389         /// </summary>
390         /// <param name="first">
391         /// <para>The first.</para>
392         /// <para></para>
393         /// </param>
394         /// <param name="second">
395         /// <para>The second.</para>
396         /// <para></para>
397         /// </param>
398         /// <returns>
399         /// <para>The bool</para>
400         /// <para></para>
401         /// </returns>
402         [MethodImpl(MethodImplOptions.AggressiveInlining)]
403         protected override bool FirstIsToTheLeftOfSecond(TLinkAddress first, TLinkAddress
            ↪ second) => GetKeyPartValue(first) < GetKeyPartValue(second);
404
405         /// <summary>
```

```
406         /// <para>
407         /// Determines whether this instance first is to the right of second.
408         /// </para>
409         /// <para></para>
410         /// </summary>
411         /// <param name="first">
412         /// <para>The first.</para>
413         /// <para></para>
414         /// </param>
415         /// <param name="second">
416         /// <para>The second.</para>
417         /// <para></para>
418         /// </param>
419         /// <returns>
420         /// <para>The bool</para>
421         /// <para></para>
422         /// </returns>
423         [MethodImpl(MethodImplOptions.AggressiveInlining)]
424         protected override bool FirstIsToTheRightOfSecond(TLinkAddress first, TLinkAddress
            ↪ second) => GetKeyPartValue(first) > GetKeyPartValue(second);
425     }
426 }
```

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

```
1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.Memory.Split.Generic;
3  using TLinkAddress = System.UInt32;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Memory.Split.Specific
8  {
9      /// <summary>
10     /// <para>
11     /// Represents the int 32 internal links size balanced tree methods base.
12     /// </para>
13     /// <para></para>
14     /// </summary>
15     /// <seealso cref="InternalLinksSizeBalancedTreeMethodsBase{TLinkAddress}"/>
16     public unsafe abstract class UInt32InternalLinksSizeBalancedTreeMethodsBase :
       ↪ InternalLinksSizeBalancedTreeMethodsBase<TLinkAddress>
17     {
18         /// <summary>
19         /// <para>
20         /// The links data parts.
21         /// </para>
22         /// <para></para>
23         /// </summary>
24         protected new readonly RawLinkDataPart<TLinkAddress>* LinksDataParts;
25         /// <summary>
26         /// <para>
27         /// The links index parts.
28         /// </para>
29         /// <para></para>
30         /// </summary>
31         protected new readonly RawLinkIndexPart<TLinkAddress>* LinksIndexParts;
32         /// <summary>
33         /// <para>
34         /// The header.
35         /// </para>
36         /// <para></para>
37         /// </summary>
38         protected new readonly LinksHeader<TLinkAddress>* Header;
39
40         /// <summary>
41         /// <para>
42         /// Initializes a new <see cref="UInt32InternalLinksSizeBalancedTreeMethodsBase"/>
            ↪ instance.
43         /// </para>
44         /// <para></para>
45         /// </summary>
46         /// <param name="constants">
47         /// <para>A constants.</para>
48         /// <para></para>
49         /// </param>
50         /// <param name="linksDataParts">
51         /// <para>A links data parts.</para>
52         /// <para></para>
```

```
        /// </param>
        /// <param name="linksIndexParts">
        /// <para>A links index parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected UInt32InternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLinkAddress>
        ↪  constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
        ↪  RawLinkIndexPart<TLinkAddress>* linksIndexParts, LinksHeader<TLinkAddress>* header)
            : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
        {
            LinksDataParts = linksDataParts;
            LinksIndexParts = linksIndexParts;
            Header = header;
        }

        /// <summary>
        /// <para>
        /// Gets the zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetZero() => 0U;

        /// <summary>
        /// <para>
        /// Determines whether this instance equal to zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool EqualToZero(TLinkAddress value) => value == 0U;

        /// <summary>
        /// <para>
        /// Determines whether this instance are equal.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool AreEqual(TLinkAddress first, TLinkAddress second) => first ==
        ↪  second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater than zero.
        /// </para>
        /// <para></para>
        /// </summary>
```

```
128         /// <param name="value">
129         /// <para>The value.</para>
130         /// <para></para>
131         /// </param>
132         /// <returns>
133         /// <para>The bool</para>
134         /// <para></para>
135         /// </returns>
136         [MethodImpl(MethodImplOptions.AggressiveInlining)]
137         protected override bool GreaterThanZero(TLinkAddress value) => value > 0U;
138
139         /// <summary>
140         /// <para>
141         /// Determines whether this instance greater than.
142         /// </para>
143         /// <para></para>
144         /// </summary>
145         /// <param name="first">
146         /// <para>The first.</para>
147         /// <para></para>
148         /// </param>
149         /// <param name="second">
150         /// <para>The second.</para>
151         /// <para></para>
152         /// </param>
153         /// <returns>
154         /// <para>The bool</para>
155         /// <para></para>
156         /// </returns>
157         [MethodImpl(MethodImplOptions.AggressiveInlining)]
158         protected override bool GreaterThan(TLinkAddress first, TLinkAddress second) => first >
          ↪  second;
159
160         /// <summary>
161         /// <para>
162         /// Determines whether this instance greater or equal than.
163         /// </para>
164         /// <para></para>
165         /// </summary>
166         /// <param name="first">
167         /// <para>The first.</para>
168         /// <para></para>
169         /// </param>
170         /// <param name="second">
171         /// <para>The second.</para>
172         /// <para></para>
173         /// </param>
174         /// <returns>
175         /// <para>The bool</para>
176         /// <para></para>
177         /// </returns>
178         [MethodImpl(MethodImplOptions.AggressiveInlining)]
179         protected override bool GreaterOrEqualThan(TLinkAddress first, TLinkAddress second) =>
          ↪  first >= second;
180
181         /// <summary>
182         /// <para>
183         /// Determines whether this instance greater or equal than zero.
184         /// </para>
185         /// <para></para>
186         /// </summary>
187         /// <param name="value">
188         /// <para>The value.</para>
189         /// <para></para>
190         /// </param>
191         /// <returns>
192         /// <para>The bool</para>
193         /// <para></para>
194         /// </returns>
195         [MethodImpl(MethodImplOptions.AggressiveInlining)]
196         protected override bool GreaterOrEqualThanZero(TLinkAddress value) => true; // value >=
          ↪  0 is always true for ulong
197
198         /// <summary>
199         /// <para>
200         /// Determines whether this instance less or equal than zero.
201         /// </para>
202         /// <para></para>
```

```
203         /// </summary>
204         /// <param name="value">
205         /// <para>The value.</para>
206         /// <para></para>
207         /// </param>
208         /// <returns>
209         /// <para>The bool</para>
210         /// <para></para>
211         /// </returns>
212         [MethodImpl(MethodImplOptions.AggressiveInlining)]
213         protected override bool LessOrEqualThanZero(TLinkAddress value) => value == OUL; //
            ↪  value is always >= 0 for ulong
214
215         /// <summary>
216         /// <para>
217         /// Determines whether this instance less or equal than.
218         /// </para>
219         /// <para></para>
220         /// </summary>
221         /// <param name="first">
222         /// <para>The first.</para>
223         /// <para></para>
224         /// </param>
225         /// <param name="second">
226         /// <para>The second.</para>
227         /// <para></para>
228         /// </param>
229         /// <returns>
230         /// <para>The bool</para>
231         /// <para></para>
232         /// </returns>
233         [MethodImpl(MethodImplOptions.AggressiveInlining)]
234         protected override bool LessOrEqualThan(TLinkAddress first, TLinkAddress second) =>
            ↪  first <= second;
235
236         /// <summary>
237         /// <para>
238         /// Determines whether this instance less than zero.
239         /// </para>
240         /// <para></para>
241         /// </summary>
242         /// <param name="value">
243         /// <para>The value.</para>
244         /// <para></para>
245         /// </param>
246         /// <returns>
247         /// <para>The bool</para>
248         /// <para></para>
249         /// </returns>
250         [MethodImpl(MethodImplOptions.AggressiveInlining)]
251         protected override bool LessThanZero(TLinkAddress value) => false; // value < 0 is
            ↪  always false for ulong
252
253         /// <summary>
254         /// <para>
255         /// Determines whether this instance less than.
256         /// </para>
257         /// <para></para>
258         /// </summary>
259         /// <param name="first">
260         /// <para>The first.</para>
261         /// <para></para>
262         /// </param>
263         /// <param name="second">
264         /// <para>The second.</para>
265         /// <para></para>
266         /// </param>
267         /// <returns>
268         /// <para>The bool</para>
269         /// <para></para>
270         /// </returns>
271         [MethodImpl(MethodImplOptions.AggressiveInlining)]
272         protected override bool LessThan(TLinkAddress first, TLinkAddress second) => first <
            ↪  second;
273
274         /// <summary>
275         /// <para>
276         /// Increments the value.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress Increment(TLinkAddress value) => ++value;

        /// <summary>
        /// <para>
        /// Decrements the value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress Decrement(TLinkAddress value) => --value;

        /// <summary>
        /// <para>
        /// Adds the first.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress Add(TLinkAddress first, TLinkAddress second) => first +
        ↪  second;

        /// <summary>
        /// <para>
        /// Subtracts the first.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress Subtract(TLinkAddress first, TLinkAddress second) =>
        ↪  first - second;

        /// <summary>
        /// <para>
        /// Gets the link data part reference using the specified link.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link data part of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkDataPart<TLinkAddress>
        ↪  GetLinkDataPartReference(TLinkAddress link) => ref LinksDataParts[link];

        /// <summary>
        /// <para>
        /// Gets the link index part reference using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link index part of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkIndexPart<TLinkAddress>
        ↪  GetLinkIndexPartReference(TLinkAddress link) => ref LinksIndexParts[link];

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLinkAddress first, TLinkAddress
        ↪  second) => GetKeyPartValue(first) < GetKeyPartValue(second);

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the right of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLinkAddress first, TLinkAddress
        ↪  second) => GetKeyPartValue(first) > GetKeyPartValue(second);
    }
}
```

```
using System.Runtime.CompilerServices;
using TLinkAddress = System.UInt32;

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

namespace Platform.Data.Doublets.Memory.Split.Generic
{
    /// <summary>
    /// <para>
    /// Represents the int 32 internal links sources linked list methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="InternalLinksSourcesLinkedListMethods{TLinkAddress}"/>
    public unsafe class UInt32InternalLinksSourcesLinkedListMethods :
    ↪ InternalLinksSourcesLinkedListMethods<TLinkAddress>
    {
        private readonly RawLinkDataPart<TLinkAddress>* _linksDataParts;
        private readonly RawLinkIndexPart<TLinkAddress>* _linksIndexParts;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt32InternalLinksSourcesLinkedListMethods"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksDataParts">
        /// <para>A links data parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksIndexParts">
        /// <para>A links index parts.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32InternalLinksSourcesLinkedListMethods(LinksConstants<TLinkAddress>
        ↪ constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
        ↪ RawLinkIndexPart<TLinkAddress>* linksIndexParts)
            : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts)
        {
            _linksDataParts = linksDataParts;
            _linksIndexParts = linksIndexParts;
        }

        /// <summary>
        /// <para>
        /// Gets the link data part reference using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link data part of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkDataPart<TLinkAddress>
        ↪ GetLinkDataPartReference(TLinkAddress link) => ref _linksDataParts[link];

        /// <summary>
        /// <para>
        /// Gets the link index part reference using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link index part of t link</para>
```

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

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

```
using System.Runtime.CompilerServices;
using TLinkAddress = System.UInt32;

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    /// <summary>
    /// <para>
    /// Represents the int 32 internal links sources recursionless size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="UInt32InternalLinksRecursionlessSizeBalancedTreeMethodsBase"/>
    public unsafe class UInt32InternalLinksSourcesRecursionlessSizeBalancedTreeMethods :
    ↪ UInt32InternalLinksRecursionlessSizeBalancedTreeMethodsBase
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see
        ↪ cref="UInt32InternalLinksSourcesRecursionlessSizeBalancedTreeMethods"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksDataParts">
        /// <para>A links data parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksIndexParts">
        /// <para>A links index parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<TLi
        ↪ nkAddress> constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
        ↪ RawLinkIndexPart<TLinkAddress>* linksIndexParts, LinksHeader<TLinkAddress>* header)
        ↪ : base(constants, linksDataParts, linksIndexParts, header) { }

        /// <summary>
        /// <para>
        /// Gets the left reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
        ↪ LinksIndexParts[node].LeftAsSource;

        /// <summary>
        /// <para>
        /// Gets the right reference using the specified node.
        /// </para>
        /// <para></para>
```

```
64          /// </summary>
65          /// <param name="node">
66          /// <para>The node.</para>
67          /// <para></para>
68          /// </param>
69          /// <returns>
70          /// <para>The ref link</para>
71          /// <para></para>
72          /// </returns>
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
            ↪ LinksIndexParts[node].RightAsSource;
75
76          /// <summary>
77          /// <para>
78          /// Gets the left using the specified node.
79          /// </para>
80          /// <para></para>
81          /// </summary>
82          /// <param name="node">
83          /// <para>The node.</para>
84          /// <para></para>
85          /// </param>
86          /// <returns>
87          /// <para>The link</para>
88          /// <para></para>
89          /// </returns>
90          [MethodImpl(MethodImplOptions.AggressiveInlining)]
91          protected override TLinkAddress GetLeft(TLinkAddress node) =>
            ↪ LinksIndexParts[node].LeftAsSource;
92
93          /// <summary>
94          /// <para>
95          /// Gets the right using the specified node.
96          /// </para>
97          /// <para></para>
98          /// </summary>
99          /// <param name="node">
100         /// <para>The node.</para>
101         /// <para></para>
102         /// </param>
103         /// <returns>
104         /// <para>The link</para>
105         /// <para></para>
106         /// </returns>
107         [MethodImpl(MethodImplOptions.AggressiveInlining)]
108         protected override TLinkAddress GetRight(TLinkAddress node) =>
            ↪ LinksIndexParts[node].RightAsSource;
109
110         /// <summary>
111         /// <para>
112         /// Sets the left using the specified node.
113         /// </para>
114         /// <para></para>
115         /// </summary>
116         /// <param name="node">
117         /// <para>The node.</para>
118         /// <para></para>
119         /// </param>
120         /// <param name="left">
121         /// <para>The left.</para>
122         /// <para></para>
123         /// </param>
124         [MethodImpl(MethodImplOptions.AggressiveInlining)]
125         protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
            ↪ LinksIndexParts[node].LeftAsSource = left;
126
127         /// <summary>
128         /// <para>
129         /// Sets the right using the specified node.
130         /// </para>
131         /// <para></para>
132         /// </summary>
133         /// <param name="node">
134         /// <para>The node.</para>
135         /// <para></para>
136         /// </param>
137         /// <param name="right">
```

```
        /// <para>The right.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
        ↪ LinksIndexParts[node].RightAsSource = right;

        /// <summary>
        /// <para>
        /// Gets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetSize(TLinkAddress node) =>
        ↪ LinksIndexParts[node].SizeAsSource;

        /// <summary>
        /// <para>
        /// Sets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="size">
        /// <para>The size.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
        ↪ LinksIndexParts[node].SizeAsSource = size;

        /// <summary>
        /// <para>
        /// Gets the tree root using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetTreeRoot(TLinkAddress node) =>
        ↪ LinksIndexParts[node].RootAsSource;

        /// <summary>
        /// <para>
        /// Gets the base part value using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetBasePartValue(TLinkAddress node) =>
        ↪ LinksDataParts[node].Source;
```

```
211
212         /// <summary>
213         /// <para>
214         /// Gets the key part value using the specified node.
215         /// </para>
216         /// <para></para>
217         /// </summary>
218         /// <param name="node">
219         /// <para>The node.</para>
220         /// <para></para>
221         /// </param>
222         /// <returns>
223         /// <para>The link</para>
224         /// <para></para>
225         /// </returns>
226         [MethodImpl(MethodImplOptions.AggressiveInlining)]
227         protected override TLinkAddress GetKeyPartValue(TLinkAddress node) =>
            ↪ LinksDataParts[node].Target;
228
229         /// <summary>
230         /// <para>
231         /// Clears the node using the specified node.
232         /// </para>
233         /// <para></para>
234         /// </summary>
235         /// <param name="node">
236         /// <para>The node.</para>
237         /// <para></para>
238         /// </param>
239         [MethodImpl(MethodImplOptions.AggressiveInlining)]
240         protected override void ClearNode(TLinkAddress node)
241         {
242             ref var link = ref LinksIndexParts[node];
243             link.LeftAsSource = Zero;
244             link.RightAsSource = Zero;
245             link.SizeAsSource = Zero;
246         }
247
248         /// <summary>
249         /// <para>
250         /// Searches the source.
251         /// </para>
252         /// <para></para>
253         /// </summary>
254         /// <param name="source">
255         /// <para>The source.</para>
256         /// <para></para>
257         /// </param>
258         /// <param name="target">
259         /// <para>The target.</para>
260         /// <para></para>
261         /// </param>
262         /// <returns>
263         /// <para>The link</para>
264         /// <para></para>
265         /// </returns>
266         public override TLinkAddress Search(TLinkAddress source, TLinkAddress target) =>
            ↪ SearchCore(GetTreeRoot(source), target);
267     }
268 }
```

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

```
1  using System.Runtime.CompilerServices;
2  using TLinkAddress = System.UInt32;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.Split.Specific
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the int 32 internal links sources size balanced tree methods.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     /// <seealso cref="UInt32InternalLinksSizeBalancedTreeMethodsBase"/>
15     public unsafe class UInt32InternalLinksSourcesSizeBalancedTreeMethods :
        ↪ UInt32InternalLinksSizeBalancedTreeMethodsBase
```

```
16      {
17          /// <summary>
18          /// <para>
19          /// Initializes a new <see cref="UInt32InternalLinksSourcesSizeBalancedTreeMethods"/>
            ↪  instance.
20          /// </para>
21          /// <para></para>
22          /// </summary>
23          /// <param name="constants">
24          /// <para>A constants.</para>
25          /// <para></para>
26          /// </param>
27          /// <param name="linksDataParts">
28          /// <para>A links data parts.</para>
29          /// <para></para>
30          /// </param>
31          /// <param name="linksIndexParts">
32          /// <para>A links index parts.</para>
33          /// <para></para>
34          /// </param>
35          /// <param name="header">
36          /// <para>A header.</para>
37          /// <para></para>
38          /// </param>
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          public UInt32InternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLinkAddress>
            ↪  constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
            ↪  RawLinkIndexPart<TLinkAddress>* linksIndexParts, LinksHeader<TLinkAddress>* header)
            ↪  : base(constants, linksDataParts, linksIndexParts, header) { }
41
42          /// <summary>
43          /// <para>
44          /// Gets the left reference using the specified node.
45          /// </para>
46          /// <para></para>
47          /// </summary>
48          /// <param name="node">
49          /// <para>The node.</para>
50          /// <para></para>
51          /// </param>
52          /// <returns>
53          /// <para>The ref link</para>
54          /// <para></para>
55          /// </returns>
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
            ↪  LinksIndexParts[node].LeftAsSource;
58
59          /// <summary>
60          /// <para>
61          /// Gets the right reference using the specified node.
62          /// </para>
63          /// <para></para>
64          /// </summary>
65          /// <param name="node">
66          /// <para>The node.</para>
67          /// <para></para>
68          /// </param>
69          /// <returns>
70          /// <para>The ref link</para>
71          /// <para></para>
72          /// </returns>
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
            ↪  LinksIndexParts[node].RightAsSource;
75
76          /// <summary>
77          /// <para>
78          /// Gets the left using the specified node.
79          /// </para>
80          /// <para></para>
81          /// </summary>
82          /// <param name="node">
83          /// <para>The node.</para>
84          /// <para></para>
85          /// </param>
86          /// <returns>
87          /// <para>The link</para>
```

```
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetLeft(TLinkAddress node) =>
        ↪ LinksIndexParts[node].LeftAsSource;

        /// <summary>
        /// <para>
        /// Gets the right using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetRight(TLinkAddress node) =>
        ↪ LinksIndexParts[node].RightAsSource;

        /// <summary>
        /// <para>
        /// Sets the left using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="left">
        /// <para>The left.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
        ↪ LinksIndexParts[node].LeftAsSource = left;

        /// <summary>
        /// <para>
        /// Sets the right using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="right">
        /// <para>The right.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
        ↪ LinksIndexParts[node].RightAsSource = right;

        /// <summary>
        /// <para>
        /// Gets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetSize(TLinkAddress node) =>
        ↪ LinksIndexParts[node].SizeAsSource;
```

```
160
161         /// <summary>
162         /// <para>
163         /// Sets the size using the specified node.
164         /// </para>
165         /// <para></para>
166         /// </summary>
167         /// <param name="node">
168         /// <para>The node.</para>
169         /// <para></para>
170         /// </param>
171         /// <param name="size">
172         /// <para>The size.</para>
173         /// <para></para>
174         /// </param>
175         [MethodImpl(MethodImplOptions.AggressiveInlining)]
176         protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
            ↪ LinksIndexParts[node].SizeAsSource = size;
177
178         /// <summary>
179         /// <para>
180         /// Gets the tree root using the specified node.
181         /// </para>
182         /// <para></para>
183         /// </summary>
184         /// <param name="node">
185         /// <para>The node.</para>
186         /// <para></para>
187         /// </param>
188         /// <returns>
189         /// <para>The link</para>
190         /// <para></para>
191         /// </returns>
192         [MethodImpl(MethodImplOptions.AggressiveInlining)]
193         protected override TLinkAddress GetTreeRoot(TLinkAddress node) =>
            ↪ LinksIndexParts[node].RootAsSource;
194
195         /// <summary>
196         /// <para>
197         /// Gets the base part value using the specified node.
198         /// </para>
199         /// <para></para>
200         /// </summary>
201         /// <param name="node">
202         /// <para>The node.</para>
203         /// <para></para>
204         /// </param>
205         /// <returns>
206         /// <para>The link</para>
207         /// <para></para>
208         /// </returns>
209         [MethodImpl(MethodImplOptions.AggressiveInlining)]
210         protected override TLinkAddress GetBasePartValue(TLinkAddress node) =>
            ↪ LinksDataParts[node].Source;
211
212         /// <summary>
213         /// <para>
214         /// Gets the key part value using the specified node.
215         /// </para>
216         /// <para></para>
217         /// </summary>
218         /// <param name="node">
219         /// <para>The node.</para>
220         /// <para></para>
221         /// </param>
222         /// <returns>
223         /// <para>The link</para>
224         /// <para></para>
225         /// </returns>
226         [MethodImpl(MethodImplOptions.AggressiveInlining)]
227         protected override TLinkAddress GetKeyPartValue(TLinkAddress node) =>
            ↪ LinksDataParts[node].Target;
228
229         /// <summary>
230         /// <para>
231         /// Clears the node using the specified node.
232         /// </para>
233         /// <para></para>
```

```
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLinkAddress node)
        {
            ref var link = ref LinksIndexParts[node];
            link.LeftAsSource = Zero;
            link.RightAsSource = Zero;
            link.SizeAsSource = Zero;
        }

        /// <summary>
        /// <para>
        /// Searches the source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public override TLinkAddress Search(TLinkAddress source, TLinkAddress target) =>
        ↪ SearchCore(GetTreeRoot(source), target);
    }
}
```

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

```
using System.Runtime.CompilerServices;
using TLinkAddress = System.UInt32;

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    /// <summary>
    /// <para>
    /// Represents the int 32 internal links targets recursionless size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="UInt32InternalLinksRecursionlessSizeBalancedTreeMethodsBase"/>
    public unsafe class UInt32InternalLinksTargetsRecursionlessSizeBalancedTreeMethods :
    ↪ UInt32InternalLinksRecursionlessSizeBalancedTreeMethodsBase
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see
        ↪ cref="UInt32InternalLinksTargetsRecursionlessSizeBalancedTreeMethods"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksDataParts">
        /// <para>A links data parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksIndexParts">
        /// <para>A links index parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
```

```
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          public UInt32InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<TLi
            ↪  nkAddress> constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
            ↪  RawLinkIndexPart<TLinkAddress>* linksIndexParts, LinksHeader<TLinkAddress>* header)
            ↪  : base(constants, linksDataParts, linksIndexParts, header) { }
41
42          /// <summary>
43          /// <para>
44          /// Gets the left reference using the specified node.
45          /// </para>
46          /// <para></para>
47          /// </summary>
48          /// <param name="node">
49          /// <para>The node.</para>
50          /// <para></para>
51          /// </param>
52          /// <returns>
53          /// <para>The ref link</para>
54          /// <para></para>
55          /// </returns>
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
            ↪  LinksIndexParts[node].LeftAsTarget;
58
59          /// <summary>
60          /// <para>
61          /// Gets the right reference using the specified node.
62          /// </para>
63          /// <para></para>
64          /// </summary>
65          /// <param name="node">
66          /// <para>The node.</para>
67          /// <para></para>
68          /// </param>
69          /// <returns>
70          /// <para>The ref link</para>
71          /// <para></para>
72          /// </returns>
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
            ↪  LinksIndexParts[node].RightAsTarget;
75
76          /// <summary>
77          /// <para>
78          /// Gets the left using the specified node.
79          /// </para>
80          /// <para></para>
81          /// </summary>
82          /// <param name="node">
83          /// <para>The node.</para>
84          /// <para></para>
85          /// </param>
86          /// <returns>
87          /// <para>The link</para>
88          /// <para></para>
89          /// </returns>
90          [MethodImpl(MethodImplOptions.AggressiveInlining)]
91          protected override TLinkAddress GetLeft(TLinkAddress node) =>
            ↪  LinksIndexParts[node].LeftAsTarget;
92
93          /// <summary>
94          /// <para>
95          /// Gets the right using the specified node.
96          /// </para>
97          /// <para></para>
98          /// </summary>
99          /// <param name="node">
100         /// <para>The node.</para>
101         /// <para></para>
102         /// </param>
103         /// <returns>
104         /// <para>The link</para>
105         /// <para></para>
106         /// </returns>
107         [MethodImpl(MethodImplOptions.AggressiveInlining)]
108         protected override TLinkAddress GetRight(TLinkAddress node) =>
            ↪  LinksIndexParts[node].RightAsTarget;
109
```

```
110         /// <summary>
111         /// <para>
112         /// Sets the left using the specified node.
113         /// </para>
114         /// <para></para>
115         /// </summary>
116         /// <param name="node">
117         /// <para>The node.</para>
118         /// <para></para>
119         /// </param>
120         /// <param name="left">
121         /// <para>The left.</para>
122         /// <para></para>
123         /// </param>
124         [MethodImpl(MethodImplOptions.AggressiveInlining)]
125         protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
            ↪ LinksIndexParts[node].LeftAsTarget = left;
126
127         /// <summary>
128         /// <para>
129         /// Sets the right using the specified node.
130         /// </para>
131         /// <para></para>
132         /// </summary>
133         /// <param name="node">
134         /// <para>The node.</para>
135         /// <para></para>
136         /// </param>
137         /// <param name="right">
138         /// <para>The right.</para>
139         /// <para></para>
140         /// </param>
141         [MethodImpl(MethodImplOptions.AggressiveInlining)]
142         protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
            ↪ LinksIndexParts[node].RightAsTarget = right;
143
144         /// <summary>
145         /// <para>
146         /// Gets the size using the specified node.
147         /// </para>
148         /// <para></para>
149         /// </summary>
150         /// <param name="node">
151         /// <para>The node.</para>
152         /// <para></para>
153         /// </param>
154         /// <returns>
155         /// <para>The link</para>
156         /// <para></para>
157         /// </returns>
158         [MethodImpl(MethodImplOptions.AggressiveInlining)]
159         protected override TLinkAddress GetSize(TLinkAddress node) =>
            ↪ LinksIndexParts[node].SizeAsTarget;
160
161         /// <summary>
162         /// <para>
163         /// Sets the size using the specified node.
164         /// </para>
165         /// <para></para>
166         /// </summary>
167         /// <param name="node">
168         /// <para>The node.</para>
169         /// <para></para>
170         /// </param>
171         /// <param name="size">
172         /// <para>The size.</para>
173         /// <para></para>
174         /// </param>
175         [MethodImpl(MethodImplOptions.AggressiveInlining)]
176         protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
            ↪ LinksIndexParts[node].SizeAsTarget = size;
177
178         /// <summary>
179         /// <para>
180         /// Gets the tree root using the specified node.
181         /// </para>
182         /// <para></para>
183         /// </summary>
```

```
184             /// <param name="node">
185             /// <para>The node.</para>
186             /// <para></para>
187             /// </param>
188             /// <returns>
189             /// <para>The link</para>
190             /// <para></para>
191             /// </returns>
192             [MethodImpl(MethodImplOptions.AggressiveInlining)]
193             protected override TLinkAddress GetTreeRoot(TLinkAddress node) =>
                ↪ LinksIndexParts[node].RootAsTarget;
194
195             /// <summary>
196             /// <para>
197             /// Gets the base part value using the specified node.
198             /// </para>
199             /// <para></para>
200             /// </summary>
201             /// <param name="node">
202             /// <para>The node.</para>
203             /// <para></para>
204             /// </param>
205             /// <returns>
206             /// <para>The link</para>
207             /// <para></para>
208             /// </returns>
209             [MethodImpl(MethodImplOptions.AggressiveInlining)]
210             protected override TLinkAddress GetBasePartValue(TLinkAddress node) =>
                ↪ LinksDataParts[node].Target;
211
212             /// <summary>
213             /// <para>
214             /// Gets the key part value using the specified node.
215             /// </para>
216             /// <para></para>
217             /// </summary>
218             /// <param name="node">
219             /// <para>The node.</para>
220             /// <para></para>
221             /// </param>
222             /// <returns>
223             /// <para>The link</para>
224             /// <para></para>
225             /// </returns>
226             [MethodImpl(MethodImplOptions.AggressiveInlining)]
227             protected override TLinkAddress GetKeyPartValue(TLinkAddress node) =>
                ↪ LinksDataParts[node].Source;
228
229             /// <summary>
230             /// <para>
231             /// Clears the node using the specified node.
232             /// </para>
233             /// <para></para>
234             /// </summary>
235             /// <param name="node">
236             /// <para>The node.</para>
237             /// <para></para>
238             /// </param>
239             [MethodImpl(MethodImplOptions.AggressiveInlining)]
240             protected override void ClearNode(TLinkAddress node)
241             {
242                 ref var link = ref LinksIndexParts[node];
243                 link.LeftAsTarget = Zero;
244                 link.RightAsTarget = Zero;
245                 link.SizeAsTarget = Zero;
246             }
247
248             /// <summary>
249             /// <para>
250             /// Searches the source.
251             /// </para>
252             /// <para></para>
253             /// </summary>
254             /// <param name="source">
255             /// <para>The source.</para>
256             /// <para></para>
257             /// </param>
258             /// <param name="target">
```

```
259         /// <para>The target.</para>
260         /// <para></para>
261         /// </param>
262         /// <returns>
263         /// <para>The link</para>
264         /// <para></para>
265         /// </returns>
266         public override TLinkAddress Search(TLinkAddress source, TLinkAddress target) =>
            ↪ SearchCore(GetTreeRoot(target), source);
267     }
268 }
```

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

```
1  using System.Runtime.CompilerServices;
2  using TLinkAddress = System.UInt32;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.Split.Specific
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the int 32 internal links targets size balanced tree methods.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     /// <seealso cref="UInt32InternalLinksSizeBalancedTreeMethodsBase"/>
15     public unsafe class UInt32InternalLinksTargetsSizeBalancedTreeMethods :
        ↪ UInt32InternalLinksSizeBalancedTreeMethodsBase
16     {
17         /// <summary>
18         /// <para>
19         /// Initializes a new <see cref="UInt32InternalLinksTargetsSizeBalancedTreeMethods"/>
            ↪ instance.
20         /// </para>
21         /// <para></para>
22         /// </summary>
23         /// <param name="constants">
24         /// <para>A constants.</para>
25         /// <para></para>
26         /// </param>
27         /// <param name="linksDataParts">
28         /// <para>A links data parts.</para>
29         /// <para></para>
30         /// </param>
31         /// <param name="linksIndexParts">
32         /// <para>A links index parts.</para>
33         /// <para></para>
34         /// </param>
35         /// <param name="header">
36         /// <para>A header.</para>
37         /// <para></para>
38         /// </param>
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         public UInt32InternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLinkAddress>
            ↪ constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
            ↪ RawLinkIndexPart<TLinkAddress>* linksIndexParts, LinksHeader<TLinkAddress>* header)
            ↪ : base(constants, linksDataParts, linksIndexParts, header) { }
41
42         /// <summary>
43         /// <para>
44         /// Gets the left reference using the specified node.
45         /// </para>
46         /// <para></para>
47         /// </summary>
48         /// <param name="node">
49         /// <para>The node.</para>
50         /// <para></para>
51         /// </param>
52         /// <returns>
53         /// <para>The ref link</para>
54         /// <para></para>
55         /// </returns>
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
            ↪ LinksIndexParts[node].LeftAsTarget;
58
59         /// <summary>
```

```
60          /// <para>
61          /// Gets the right reference using the specified node.
62          /// </para>
63          /// <para></para>
64          /// </summary>
65          /// <param name="node">
66          /// <para>The node.</para>
67          /// <para></para>
68          /// </param>
69          /// <returns>
70          /// <para>The ref link</para>
71          /// <para></para>
72          /// </returns>
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
            ↪ LinksIndexParts[node].RightAsTarget;
75
76          /// <summary>
77          /// <para>
78          /// Gets the left using the specified node.
79          /// </para>
80          /// <para></para>
81          /// </summary>
82          /// <param name="node">
83          /// <para>The node.</para>
84          /// <para></para>
85          /// </param>
86          /// <returns>
87          /// <para>The link</para>
88          /// <para></para>
89          /// </returns>
90          [MethodImpl(MethodImplOptions.AggressiveInlining)]
91          protected override TLinkAddress GetLeft(TLinkAddress node) =>
            ↪ LinksIndexParts[node].LeftAsTarget;
92
93          /// <summary>
94          /// <para>
95          /// Gets the right using the specified node.
96          /// </para>
97          /// <para></para>
98          /// </summary>
99          /// <param name="node">
100         /// <para>The node.</para>
101         /// <para></para>
102         /// </param>
103         /// <returns>
104         /// <para>The link</para>
105         /// <para></para>
106         /// </returns>
107         [MethodImpl(MethodImplOptions.AggressiveInlining)]
108         protected override TLinkAddress GetRight(TLinkAddress node) =>
            ↪ LinksIndexParts[node].RightAsTarget;
109
110         /// <summary>
111         /// <para>
112         /// Sets the left using the specified node.
113         /// </para>
114         /// <para></para>
115         /// </summary>
116         /// <param name="node">
117         /// <para>The node.</para>
118         /// <para></para>
119         /// </param>
120         /// <param name="left">
121         /// <para>The left.</para>
122         /// <para></para>
123         /// </param>
124         [MethodImpl(MethodImplOptions.AggressiveInlining)]
125         protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
            ↪ LinksIndexParts[node].LeftAsTarget = left;
126
127         /// <summary>
128         /// <para>
129         /// Sets the right using the specified node.
130         /// </para>
131         /// <para></para>
132         /// </summary>
133         /// <param name="node">
```

```
134             /// <para>The node.</para>
135             /// <para></para>
136             /// </param>
137             /// <param name="right">
138             /// <para>The right.</para>
139             /// <para></para>
140             /// </param>
141             [MethodImpl(MethodImplOptions.AggressiveInlining)]
142             protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
                ↪ LinksIndexParts[node].RightAsTarget = right;
143
144             /// <summary>
145             /// <para>
146             /// Gets the size using the specified node.
147             /// </para>
148             /// <para></para>
149             /// </summary>
150             /// <param name="node">
151             /// <para>The node.</para>
152             /// <para></para>
153             /// </param>
154             /// <returns>
155             /// <para>The link</para>
156             /// <para></para>
157             /// </returns>
158             [MethodImpl(MethodImplOptions.AggressiveInlining)]
159             protected override TLinkAddress GetSize(TLinkAddress node) =>
                ↪ LinksIndexParts[node].SizeAsTarget;
160
161             /// <summary>
162             /// <para>
163             /// Sets the size using the specified node.
164             /// </para>
165             /// <para></para>
166             /// </summary>
167             /// <param name="node">
168             /// <para>The node.</para>
169             /// <para></para>
170             /// </param>
171             /// <param name="size">
172             /// <para>The size.</para>
173             /// <para></para>
174             /// </param>
175             [MethodImpl(MethodImplOptions.AggressiveInlining)]
176             protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
                ↪ LinksIndexParts[node].SizeAsTarget = size;
177
178             /// <summary>
179             /// <para>
180             /// Gets the tree root using the specified node.
181             /// </para>
182             /// <para></para>
183             /// </summary>
184             /// <param name="node">
185             /// <para>The node.</para>
186             /// <para></para>
187             /// </param>
188             /// <returns>
189             /// <para>The link</para>
190             /// <para></para>
191             /// </returns>
192             [MethodImpl(MethodImplOptions.AggressiveInlining)]
193             protected override TLinkAddress GetTreeRoot(TLinkAddress node) =>
                ↪ LinksIndexParts[node].RootAsTarget;
194
195             /// <summary>
196             /// <para>
197             /// Gets the base part value using the specified node.
198             /// </para>
199             /// <para></para>
200             /// </summary>
201             /// <param name="node">
202             /// <para>The node.</para>
203             /// <para></para>
204             /// </param>
205             /// <returns>
206             /// <para>The link</para>
207             /// <para></para>
```

```
208             /// </returns>
209             [MethodImpl(MethodImplOptions.AggressiveInlining)]
210             protected override TLinkAddress GetBasePartValue(TLinkAddress node) =>
                ↪ LinksDataParts[node].Target;
211
212             /// <summary>
213             /// <para>
214             /// Gets the key part value using the specified node.
215             /// </para>
216             /// <para></para>
217             /// </summary>
218             /// <param name="node">
219             /// <para>The node.</para>
220             /// <para></para>
221             /// </param>
222             /// <returns>
223             /// <para>The link</para>
224             /// <para></para>
225             /// </returns>
226             [MethodImpl(MethodImplOptions.AggressiveInlining)]
227             protected override TLinkAddress GetKeyPartValue(TLinkAddress node) =>
                ↪ LinksDataParts[node].Source;
228
229             /// <summary>
230             /// <para>
231             /// Clears the node using the specified node.
232             /// </para>
233             /// <para></para>
234             /// </summary>
235             /// <param name="node">
236             /// <para>The node.</para>
237             /// <para></para>
238             /// </param>
239             [MethodImpl(MethodImplOptions.AggressiveInlining)]
240             protected override void ClearNode(TLinkAddress node)
241             {
242                 ref var link = ref LinksIndexParts[node];
243                 link.LeftAsTarget = Zero;
244                 link.RightAsTarget = Zero;
245                 link.SizeAsTarget = Zero;
246             }
247
248             /// <summary>
249             /// <para>
250             /// Searches the source.
251             /// </para>
252             /// <para></para>
253             /// </summary>
254             /// <param name="source">
255             /// <para>The source.</para>
256             /// <para></para>
257             /// </param>
258             /// <param name="target">
259             /// <para>The target.</para>
260             /// <para></para>
261             /// </param>
262             /// <returns>
263             /// <para>The link</para>
264             /// <para></para>
265             /// </returns>
266             public override TLinkAddress Search(TLinkAddress source, TLinkAddress target) =>
                ↪ SearchCore(GetTreeRoot(target), source);
267         }
268     }
```

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

```
1   using System;
2   using System.Runtime.CompilerServices;
3   using Platform.Singletons;
4   using Platform.Memory;
5   using Platform.Data.Doublets.Memory.Split.Generic;
6   using TLinkAddress = System.UInt32;
7
8   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10  namespace Platform.Data.Doublets.Memory.Split.Specific
11  {
12      /// <summary>
13      /// <para>
```

```
14      /// Represents the int 32 split memory links.
15      /// </para>
16      /// <para></para>
17      /// </summary>
18      /// <seealso cref="SplitMemoryLinksBase{TLinkAddress}"/>
19      public unsafe class UInt32SplitMemoryLinks : SplitMemoryLinksBase<TLinkAddress>
20      {
21          private readonly Func<ILinksTreeMethods<TLinkAddress>> _createInternalSourceTreeMethods;
22          private readonly Func<ILinksTreeMethods<TLinkAddress>> _createExternalSourceTreeMethods;
23          private readonly Func<ILinksTreeMethods<TLinkAddress>> _createInternalTargetTreeMethods;
24          private readonly Func<ILinksTreeMethods<TLinkAddress>> _createExternalTargetTreeMethods;
25          private LinksHeader<TLinkAddress>* _header;
26          private RawLinkDataPart<TLinkAddress>* _linksDataParts;
27          private RawLinkIndexPart<TLinkAddress>* _linksIndexParts;
28
29          /// <summary>
30          /// <para>
31          /// Initializes a new <see cref="UInt32SplitMemoryLinks"/> instance.
32          /// </para>
33          /// <para></para>
34          /// </summary>
35          /// <param name="dataMemory">
36          /// <para>A data memory.</para>
37          /// <para></para>
38          /// </param>
39          /// <param name="indexMemory">
40          /// <para>A index memory.</para>
41          /// <para></para>
42          /// </param>
43          [MethodImpl(MethodImplOptions.AggressiveInlining)]
44          public UInt32SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪  indexMemory) : this(dataMemory, indexMemory, DefaultLinksSizeStep) { }
45
46          /// <summary>
47          /// <para>
48          /// Initializes a new <see cref="UInt32SplitMemoryLinks"/> instance.
49          /// </para>
50          /// <para></para>
51          /// </summary>
52          /// <param name="dataMemory">
53          /// <para>A data memory.</para>
54          /// <para></para>
55          /// </param>
56          /// <param name="indexMemory">
57          /// <para>A index memory.</para>
58          /// <para></para>
59          /// </param>
60          /// <param name="memoryReservationStep">
61          /// <para>A memory reservation step.</para>
62          /// <para></para>
63          /// </param>
64          [MethodImpl(MethodImplOptions.AggressiveInlining)]
65          public UInt32SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪  indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
            ↪  memoryReservationStep, Default<LinksConstants<TLinkAddress>>.Instance,
            ↪  IndexTreeType.Default, useLinkedList: true) { }
66
67          /// <summary>
68          /// <para>
69          /// Initializes a new <see cref="UInt32SplitMemoryLinks"/> instance.
70          /// </para>
71          /// <para></para>
72          /// </summary>
73          /// <param name="dataMemory">
74          /// <para>A data memory.</para>
75          /// <para></para>
76          /// </param>
77          /// <param name="indexMemory">
78          /// <para>A index memory.</para>
79          /// <para></para>
80          /// </param>
81          /// <param name="memoryReservationStep">
82          /// <para>A memory reservation step.</para>
83          /// <para></para>
84          /// </param>
85          /// <param name="constants">
86          /// <para>A constants.</para>
87          /// <para></para>
```

```
88          /// </param>
89          [MethodImpl(MethodImplOptions.AggressiveInlining)]
90          public UInt32SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪  indexMemory, long memoryReservationStep, LinksConstants<TLinkAddress> constants) :
            ↪  this(dataMemory, indexMemory, memoryReservationStep, constants,
            ↪  IndexTreeType.Default, useLinkedList: true) { }
91
92          /// <summary>
93          /// <para>
94          /// Initializes a new <see cref="UInt32SplitMemoryLinks"/> instance.
95          /// </para>
96          /// <para></para>
97          /// </summary>
98          /// <param name="dataMemory">
99          /// <para>A data memory.</para>
100         /// <para></para>
101         /// </param>
102         /// <param name="indexMemory">
103         /// <para>A index memory.</para>
104         /// <para></para>
105         /// </param>
106         /// <param name="memoryReservationStep">
107         /// <para>A memory reservation step.</para>
108         /// <para></para>
109         /// </param>
110         /// <param name="constants">
111         /// <para>A constants.</para>
112         /// <para></para>
113         /// </param>
114         /// <param name="indexTreeType">
115         /// <para>A index tree type.</para>
116         /// <para></para>
117         /// </param>
118         /// <param name="useLinkedList">
119         /// <para>A use linked list.</para>
120         /// <para></para>
121         /// </param>
122         [MethodImpl(MethodImplOptions.AggressiveInlining)]
123         public UInt32SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪  indexMemory, long memoryReservationStep, LinksConstants<TLinkAddress> constants,
            ↪  IndexTreeType indexTreeType, bool useLinkedList) : base(dataMemory, indexMemory,
            ↪  memoryReservationStep, constants, useLinkedList)
124         {
125             if (indexTreeType == IndexTreeType.SizeBalancedTree)
126             {
127                 _createInternalSourceTreeMethods = () => new
                    ↪  UInt32InternalLinksSourcesSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
128                 _createExternalSourceTreeMethods = () => new
                    ↪  UInt32ExternalLinksSourcesSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
129                 _createInternalTargetTreeMethods = () => new
                    ↪  UInt32InternalLinksTargetsSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
130                 _createExternalTargetTreeMethods = () => new
                    ↪  UInt32ExternalLinksTargetsSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
131             }
132             else
133             {
134                 _createInternalSourceTreeMethods = () => new
                    ↪  UInt32InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
135                 _createExternalSourceTreeMethods = () => new
                    ↪  UInt32ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
136                 _createInternalTargetTreeMethods = () => new
                    ↪  UInt32InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
137                 _createExternalTargetTreeMethods = () => new
                    ↪  UInt32ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
138             }
139             Init(dataMemory, indexMemory);
140         }
141
142         /// <summary>
```

```
143             /// <para>
144             /// Sets the pointers using the specified data memory.
145             /// </para>
146             /// <para></para>
147             /// </summary>
148             /// <param name="dataMemory">
149             /// <para>The data memory.</para>
150             /// <para></para>
151             /// </param>
152             /// <param name="indexMemory">
153             /// <para>The index memory.</para>
154             /// <para></para>
155             /// </param>
156             [MethodImpl(MethodImplOptions.AggressiveInlining)]
157             protected override void SetPointers(IResizableDirectMemory dataMemory,
                ↪  IResizableDirectMemory indexMemory)
158             {
159                 _linksDataParts = (RawLinkDataPart<TLinkAddress>*)dataMemory.Pointer;
160                 _linksIndexParts = (RawLinkIndexPart<TLinkAddress>*)indexMemory.Pointer;
161                 _header = (LinksHeader<TLinkAddress>*)indexMemory.Pointer;
162                 if (_useLinkedList)
163                 {
164                     InternalSourcesListMethods = new
                        ↪  UInt32InternalLinksSourcesLinkedListMethods(Constants, _linksDataParts,
                        ↪  _linksIndexParts);
165                 }
166                 else
167                 {
168                     InternalSourcesTreeMethods = _createInternalSourceTreeMethods();
169                 }
170                 ExternalSourcesTreeMethods = _createExternalSourceTreeMethods();
171                 InternalTargetsTreeMethods = _createInternalTargetTreeMethods();
172                 ExternalTargetsTreeMethods = _createExternalTargetTreeMethods();
173                 UnusedLinksListMethods = new UInt32UnusedLinksListMethods(_linksDataParts, _header);
174             }
175 
176             /// <summary>
177             /// <para>
178             /// Resets the pointers.
179             /// </para>
180             /// <para></para>
181             /// </summary>
182             [MethodImpl(MethodImplOptions.AggressiveInlining)]
183             protected override void ResetPointers()
184             {
185                 base.ResetPointers();
186                 _linksDataParts = null;
187                 _linksIndexParts = null;
188                 _header = null;
189             }
190 
191             /// <summary>
192             /// <para>
193             /// Gets the header reference.
194             /// </para>
195             /// <para></para>
196             /// </summary>
197             /// <returns>
198             /// <para>A ref links header of t link</para>
199             /// <para></para>
200             /// </returns>
201             [MethodImpl(MethodImplOptions.AggressiveInlining)]
202             protected override ref LinksHeader<TLinkAddress> GetHeaderReference() => ref *_header;
203 
204             /// <summary>
205             /// <para>
206             /// Gets the link data part reference using the specified link index.
207             /// </para>
208             /// <para></para>
209             /// </summary>
210             /// <param name="linkIndex">
211             /// <para>The link index.</para>
212             /// <para></para>
213             /// </param>
214             /// <returns>
215             /// <para>A ref raw link data part of t link</para>
216             /// <para></para>
217             /// </returns>
```

```
218         [MethodImpl(MethodImplOptions.AggressiveInlining)]
219         protected override ref RawLinkDataPart<TLinkAddress>
            ↪ GetLinkDataPartReference(TLinkAddress linkIndex) => ref _linksDataParts[linkIndex];
220
221         /// <summary>
222         /// <para>
223         /// Gets the link index part reference using the specified link index.
224         /// </para>
225         /// <para></para>
226         /// </summary>
227         /// <param name="linkIndex">
228         /// <para>The link index.</para>
229         /// <para></para>
230         /// </param>
231         /// <returns>
232         /// <para>A ref raw link index part of t link</para>
233         /// <para></para>
234         /// </returns>
235         [MethodImpl(MethodImplOptions.AggressiveInlining)]
236         protected override ref RawLinkIndexPart<TLinkAddress>
            ↪ GetLinkIndexPartReference(TLinkAddress linkIndex) => ref _linksIndexParts[linkIndex];
237
238         /// <summary>
239         /// <para>
240         /// Determines whether this instance are equal.
241         /// </para>
242         /// <para></para>
243         /// </summary>
244         /// <param name="first">
245         /// <para>The first.</para>
246         /// <para></para>
247         /// </param>
248         /// <param name="second">
249         /// <para>The second.</para>
250         /// <para></para>
251         /// </param>
252         /// <returns>
253         /// <para>The bool</para>
254         /// <para></para>
255         /// </returns>
256         [MethodImpl(MethodImplOptions.AggressiveInlining)]
257         protected override bool AreEqual(TLinkAddress first, TLinkAddress second) => first ==
            ↪ second;
258
259         /// <summary>
260         /// <para>
261         /// Determines whether this instance less than.
262         /// </para>
263         /// <para></para>
264         /// </summary>
265         /// <param name="first">
266         /// <para>The first.</para>
267         /// <para></para>
268         /// </param>
269         /// <param name="second">
270         /// <para>The second.</para>
271         /// <para></para>
272         /// </param>
273         /// <returns>
274         /// <para>The bool</para>
275         /// <para></para>
276         /// </returns>
277         [MethodImpl(MethodImplOptions.AggressiveInlining)]
278         protected override bool LessThan(TLinkAddress first, TLinkAddress second) => first <
            ↪ second;
279
280         /// <summary>
281         /// <para>
282         /// Determines whether this instance less or equal than.
283         /// </para>
284         /// <para></para>
285         /// </summary>
286         /// <param name="first">
287         /// <para>The first.</para>
288         /// <para></para>
289         /// </param>
290         /// <param name="second">
291         /// <para>The second.</para>
```

```
292         /// <para></para>
293         /// </param>
294         /// <returns>
295         /// <para>The bool</para>
296         /// <para></para>
297         /// </returns>
298         [MethodImpl(MethodImplOptions.AggressiveInlining)]
299         protected override bool LessOrEqualThan(TLinkAddress first, TLinkAddress second) =>
            ↪ first <= second;
300
301         /// <summary>
302         /// <para>
303         /// Determines whether this instance greater than.
304         /// </para>
305         /// <para></para>
306         /// </summary>
307         /// <param name="first">
308         /// <para>The first.</para>
309         /// <para></para>
310         /// </param>
311         /// <param name="second">
312         /// <para>The second.</para>
313         /// <para></para>
314         /// </param>
315         /// <returns>
316         /// <para>The bool</para>
317         /// <para></para>
318         /// </returns>
319         [MethodImpl(MethodImplOptions.AggressiveInlining)]
320         protected override bool GreaterThan(TLinkAddress first, TLinkAddress second) => first >
            ↪ second;
321
322         /// <summary>
323         /// <para>
324         /// Determines whether this instance greater or equal than.
325         /// </para>
326         /// <para></para>
327         /// </summary>
328         /// <param name="first">
329         /// <para>The first.</para>
330         /// <para></para>
331         /// </param>
332         /// <param name="second">
333         /// <para>The second.</para>
334         /// <para></para>
335         /// </param>
336         /// <returns>
337         /// <para>The bool</para>
338         /// <para></para>
339         /// </returns>
340         [MethodImpl(MethodImplOptions.AggressiveInlining)]
341         protected override bool GreaterOrEqualThan(TLinkAddress first, TLinkAddress second) =>
            ↪ first >= second;
342
343         /// <summary>
344         /// <para>
345         /// Gets the zero.
346         /// </para>
347         /// <para></para>
348         /// </summary>
349         /// <returns>
350         /// <para>The link</para>
351         /// <para></para>
352         /// </returns>
353         [MethodImpl(MethodImplOptions.AggressiveInlining)]
354         protected override TLinkAddress GetZero() => 0U;
355
356         /// <summary>
357         /// <para>
358         /// Gets the one.
359         /// </para>
360         /// <para></para>
361         /// </summary>
362         /// <returns>
363         /// <para>The link</para>
364         /// <para></para>
365         /// </returns>
366         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
367         protected override TLinkAddress GetOne() => 1U;
368
369         /// <summary>
370         /// <para>
371         /// Converts the to int 64 using the specified value.
372         /// </para>
373         /// <para></para>
374         /// </summary>
375         /// <param name="value">
376         /// <para>The value.</para>
377         /// <para></para>
378         /// </param>
379         /// <returns>
380         /// <para>The long</para>
381         /// <para></para>
382         /// </returns>
383         [MethodImpl(MethodImplOptions.AggressiveInlining)]
384         protected override long ConvertToInt64(TLinkAddress value) => value;
385
386         /// <summary>
387         /// <para>
388         /// Converts the to address using the specified value.
389         /// </para>
390         /// <para></para>
391         /// </summary>
392         /// <param name="value">
393         /// <para>The value.</para>
394         /// <para></para>
395         /// </param>
396         /// <returns>
397         /// <para>The link</para>
398         /// <para></para>
399         /// </returns>
400         [MethodImpl(MethodImplOptions.AggressiveInlining)]
401         protected override TLinkAddress ConvertToAddress(long value) => (TLinkAddress)value;
402
403         /// <summary>
404         /// <para>
405         /// Adds the first.
406         /// </para>
407         /// <para></para>
408         /// </summary>
409         /// <param name="first">
410         /// <para>The first.</para>
411         /// <para></para>
412         /// </param>
413         /// <param name="second">
414         /// <para>The second.</para>
415         /// <para></para>
416         /// </param>
417         /// <returns>
418         /// <para>The link</para>
419         /// <para></para>
420         /// </returns>
421         [MethodImpl(MethodImplOptions.AggressiveInlining)]
422         protected override TLinkAddress Add(TLinkAddress first, TLinkAddress second) => first +
            ↪ second;
423
424         /// <summary>
425         /// <para>
426         /// Subtracts the first.
427         /// </para>
428         /// <para></para>
429         /// </summary>
430         /// <param name="first">
431         /// <para>The first.</para>
432         /// <para></para>
433         /// </param>
434         /// <param name="second">
435         /// <para>The second.</para>
436         /// <para></para>
437         /// </param>
438         /// <returns>
439         /// <para>The link</para>
440         /// <para></para>
441         /// </returns>
442         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
443         protected override TLinkAddress Subtract(TLinkAddress first, TLinkAddress second) =>
            ↪ first - second;
444
445         /// <summary>
446         /// <para>
447         /// Increments the link.
448         /// </para>
449         /// <para></para>
450         /// </summary>
451         /// <param name="link">
452         /// <para>The link.</para>
453         /// <para></para>
454         /// </param>
455         /// <returns>
456         /// <para>The link</para>
457         /// <para></para>
458         /// </returns>
459         [MethodImpl(MethodImplOptions.AggressiveInlining)]
460         protected override TLinkAddress Increment(TLinkAddress link) => ++link;
461
462         /// <summary>
463         /// <para>
464         /// Decrements the link.
465         /// </para>
466         /// <para></para>
467         /// </summary>
468         /// <param name="link">
469         /// <para>The link.</para>
470         /// <para></para>
471         /// </param>
472         /// <returns>
473         /// <para>The link</para>
474         /// <para></para>
475         /// </returns>
476         [MethodImpl(MethodImplOptions.AggressiveInlining)]
477         protected override TLinkAddress Decrement(TLinkAddress link) => --link;
478     }
479 }
```

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

```
1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.Memory.Split.Generic;
3  using TLinkAddress = System.UInt32;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Memory.Split.Specific
8  {
9      /// <summary>
10     /// <para>
11     /// Represents the int 32 unused links list methods.
12     /// </para>
13     /// <para></para>
14     /// </summary>
15     /// <seealso cref="UnusedLinksListMethods{TLinkAddress}"/>
16     public unsafe class UInt32UnusedLinksListMethods : UnusedLinksListMethods<TLinkAddress>
17     {
18         private readonly RawLinkDataPart<TLinkAddress>* _links;
19         private readonly LinksHeader<TLinkAddress>* _header;
20
21         /// <summary>
22         /// <para>
23         /// Initializes a new <see cref="UInt32UnusedLinksListMethods"/> instance.
24         /// </para>
25         /// <para></para>
26         /// </summary>
27         /// <param name="links">
28         /// <para>A links.</para>
29         /// <para></para>
30         /// </param>
31         /// <param name="header">
32         /// <para>A header.</para>
33         /// <para></para>
34         /// </param>
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         public UInt32UnusedLinksListMethods(RawLinkDataPart<TLinkAddress>* links,
            ↪ LinksHeader<TLinkAddress>* header)
37             : base((byte*)links, (byte*)header)
38         {
```

```
            _links = links;
            _header = header;
        }

        /// <summary>
        /// <para>
        /// Gets the link data part reference using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link data part of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkDataPart<TLinkAddress>
        ↪  GetLinkDataPartReference(TLinkAddress link) => ref _links[link];

        /// <summary>
        /// <para>
        /// Gets the header reference.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A ref links header of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref LinksHeader<TLinkAddress> GetHeaderReference() => ref *_header;
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    /// <summary>
    /// <para>
    /// Represents the int 64 external links recursionless size balanced tree methods base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ExternalLinksRecursionlessSizeBalancedTreeMethodsBase{TLinkAddress}"/>
    /// <seealso cref="ILinksTreeMethods{TLinkAddress}"/>
    public unsafe abstract class UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase :
    ↪  ExternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLinkAddress>,
    ↪  ILinksTreeMethods<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// The links data parts.
        /// </para>
        /// <para></para>
        /// </summary>
        protected new readonly RawLinkDataPart<TLinkAddress>* LinksDataParts;
        /// <summary>
        /// <para>
        /// The links index parts.
        /// </para>
        /// <para></para>
        /// </summary>
        protected new readonly RawLinkIndexPart<TLinkAddress>* LinksIndexParts;
        /// <summary>
        /// <para>
        /// The header.
        /// </para>
        /// <para></para>
        /// </summary>
        protected new readonly LinksHeader<TLinkAddress>* Header;
```

```
40
41          /// <summary>
42          /// <para>
43          /// Initializes a new <see
            ↪  cref="UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase"/> instance.
44          /// </para>
45          /// <para></para>
46          /// </summary>
47          /// <param name="constants">
48          /// <para>A constants.</para>
49          /// <para></para>
50          /// </param>
51          /// <param name="linksDataParts">
52          /// <para>A links data parts.</para>
53          /// <para></para>
54          /// </param>
55          /// <param name="linksIndexParts">
56          /// <para>A links index parts.</para>
57          /// <para></para>
58          /// </param>
59          /// <param name="header">
60          /// <para>A header.</para>
61          /// <para></para>
62          /// </param>
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          protected UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLi
            ↪  nkAddress> constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
            ↪  RawLinkIndexPart<TLinkAddress>* linksIndexParts, LinksHeader<TLinkAddress>* header)
65              : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
66          {
67              LinksDataParts = linksDataParts;
68              LinksIndexParts = linksIndexParts;
69              Header = header;
70          }
71
72          /// <summary>
73          /// <para>
74          /// Gets the zero.
75          /// </para>
76          /// <para></para>
77          /// </summary>
78          /// <returns>
79          /// <para>The ulong</para>
80          /// <para></para>
81          /// </returns>
82          [MethodImpl(MethodImplOptions.AggressiveInlining)]
83          protected override ulong GetZero() => 0UL;
84
85          /// <summary>
86          /// <para>
87          /// Determines whether this instance equal to zero.
88          /// </para>
89          /// <para></para>
90          /// </summary>
91          /// <param name="value">
92          /// <para>The value.</para>
93          /// <para></para>
94          /// </param>
95          /// <returns>
96          /// <para>The bool</para>
97          /// <para></para>
98          /// </returns>
99          [MethodImpl(MethodImplOptions.AggressiveInlining)]
100         protected override bool EqualToZero(ulong value) => value == 0UL;
101
102         /// <summary>
103         /// <para>
104         /// Determines whether this instance are equal.
105         /// </para>
106         /// <para></para>
107         /// </summary>
108         /// <param name="first">
109         /// <para>The first.</para>
110         /// <para></para>
111         /// </param>
112         /// <param name="second">
113         /// <para>The second.</para>
114         /// <para></para>
```

```
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool AreEqual(ulong first, ulong second) => first == second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThanZero(ulong value) => value > 0UL;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThan(ulong first, ulong second) => first > second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater or equal than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater or equal than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
```

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

        /// <summary>
        /// <para>
        /// Determines whether this instance less or equal than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
        ↪  always >= 0 for ulong

        /// <summary>
        /// <para>
        /// Determines whether this instance less or equal than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;

        /// <summary>
        /// <para>
        /// Determines whether this instance less than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
        ↪  for ulong

        /// <summary>
        /// <para>
        /// Determines whether this instance less than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
```

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

        /// <summary>
        /// <para>
        /// Increments the value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Increment(ulong value) => ++value;

        /// <summary>
        /// <para>
        /// Decrements the value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Decrement(ulong value) => --value;

        /// <summary>
        /// <para>
        /// Adds the first.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Add(ulong first, ulong second) => first + second;

        /// <summary>
        /// <para>
        /// Subtracts the first.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
```

```
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Subtract(ulong first, ulong second) => first - second;

        /// <summary>
        /// <para>
        /// Gets the header reference.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A ref links header of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref LinksHeader<TLinkAddress> GetHeaderReference() => ref *Header;

        /// <summary>
        /// <para>
        /// Gets the link data part reference using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link data part of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkDataPart<TLinkAddress>
        ↪  GetLinkDataPartReference(TLinkAddress link) => ref LinksDataParts[link];

        /// <summary>
        /// <para>
        /// Gets the link index part reference using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link index part of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkIndexPart<TLinkAddress>
        ↪  GetLinkIndexPartReference(TLinkAddress link) => ref LinksIndexParts[link];

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLinkAddress first, TLinkAddress second)
        {
            ref var firstLink = ref LinksDataParts[first];
            ref var secondLink = ref LinksDataParts[second];
```

```
421             return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
422         }
423
424         /// <summary>
425         /// <para>
426         /// Determines whether this instance first is to the right of second.
427         /// </para>
428         /// <para></para>
429         /// </summary>
430         /// <param name="first">
431         /// <para>The first.</para>
432         /// <para></para>
433         /// </param>
434         /// <param name="second">
435         /// <para>The second.</para>
436         /// <para></para>
437         /// </param>
438         /// <returns>
439         /// <para>The bool</para>
440         /// <para></para>
441         /// </returns>
442         [MethodImpl(MethodImplOptions.AggressiveInlining)]
443         protected override bool FirstIsToTheRightOfSecond(TLinkAddress first, TLinkAddress
                ↪  second)
444         {
445             ref var firstLink = ref LinksDataParts[first];
446             ref var secondLink = ref LinksDataParts[second];
447             return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
448         }
449     }
450 }
```

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

```
1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.Memory.Split.Generic;
3  using TLinkAddress = System.UInt64;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Memory.Split.Specific
8  {
9      /// <summary>
10     /// <para>
11     /// Represents the int 64 external links size balanced tree methods base.
12     /// </para>
13     /// <para></para>
14     /// </summary>
15     /// <seealso cref="ExternalLinksSizeBalancedTreeMethodsBase{TLinkAddress}"/>
16     /// <seealso cref="ILinksTreeMethods{TLinkAddress}"/>
17     public unsafe abstract class UInt64ExternalLinksSizeBalancedTreeMethodsBase :
       ↪  ExternalLinksSizeBalancedTreeMethodsBase<TLinkAddress>, ILinksTreeMethods<TLinkAddress>
18     {
19         /// <summary>
20         /// <para>
21         /// The links data parts.
22         /// </para>
23         /// <para></para>
24         /// </summary>
25         protected new readonly RawLinkDataPart<TLinkAddress>* LinksDataParts;
26         /// <summary>
27         /// <para>
28         /// The links index parts.
29         /// </para>
30         /// <para></para>
31         /// </summary>
32         protected new readonly RawLinkIndexPart<TLinkAddress>* LinksIndexParts;
33         /// <summary>
34         /// <para>
35         /// The header.
36         /// </para>
37         /// <para></para>
38         /// </summary>
39         protected new readonly LinksHeader<TLinkAddress>* Header;
40
41         /// <summary>
42         /// <para>
```

```
        /// Initializes a new <see cref="UInt64ExternalLinksSizeBalancedTreeMethodsBase"/>
        ↪ instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksDataParts">
        /// <para>A links data parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksIndexParts">
        /// <para>A links index parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected UInt64ExternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLinkAddress>
        ↪ constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
        ↪ RawLinkIndexPart<TLinkAddress>* linksIndexParts, LinksHeader<TLinkAddress>* header)
            : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
        {
            LinksDataParts = linksDataParts;
            LinksIndexParts = linksIndexParts;
            Header = header;
        }

        /// <summary>
        /// <para>
        /// Gets the zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetZero() => 0UL;

        /// <summary>
        /// <para>
        /// Determines whether this instance equal to zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool EqualToZero(ulong value) => value == 0UL;

        /// <summary>
        /// <para>
        /// Determines whether this instance are equal.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
```

```
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool AreEqual(ulong first, ulong second) => first == second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThanZero(ulong value) => value > OUL;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThan(ulong first, ulong second) => first > second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater or equal than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater or equal than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
```

```
196             [MethodImpl(MethodImplOptions.AggressiveInlining)]
197             protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
                ↪  always true for ulong
198
199             /// <summary>
200             /// <para>
201             /// Determines whether this instance less or equal than zero.
202             /// </para>
203             /// <para></para>
204             /// </summary>
205             /// <param name="value">
206             /// <para>The value.</para>
207             /// <para></para>
208             /// </param>
209             /// <returns>
210             /// <para>The bool</para>
211             /// <para></para>
212             /// </returns>
213             [MethodImpl(MethodImplOptions.AggressiveInlining)]
214             protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
                ↪  always >= 0 for ulong
215
216             /// <summary>
217             /// <para>
218             /// Determines whether this instance less or equal than.
219             /// </para>
220             /// <para></para>
221             /// </summary>
222             /// <param name="first">
223             /// <para>The first.</para>
224             /// <para></para>
225             /// </param>
226             /// <param name="second">
227             /// <para>The second.</para>
228             /// <para></para>
229             /// </param>
230             /// <returns>
231             /// <para>The bool</para>
232             /// <para></para>
233             /// </returns>
234             [MethodImpl(MethodImplOptions.AggressiveInlining)]
235             protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
236
237             /// <summary>
238             /// <para>
239             /// Determines whether this instance less than zero.
240             /// </para>
241             /// <para></para>
242             /// </summary>
243             /// <param name="value">
244             /// <para>The value.</para>
245             /// <para></para>
246             /// </param>
247             /// <returns>
248             /// <para>The bool</para>
249             /// <para></para>
250             /// </returns>
251             [MethodImpl(MethodImplOptions.AggressiveInlining)]
252             protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
                ↪  for ulong
253
254             /// <summary>
255             /// <para>
256             /// Determines whether this instance less than.
257             /// </para>
258             /// <para></para>
259             /// </summary>
260             /// <param name="first">
261             /// <para>The first.</para>
262             /// <para></para>
263             /// </param>
264             /// <param name="second">
265             /// <para>The second.</para>
266             /// <para></para>
267             /// </param>
268             /// <returns>
269             /// <para>The bool</para>
270             /// <para></para>
```

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

        /// <summary>
        /// <para>
        /// Increments the value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Increment(ulong value) => ++value;

        /// <summary>
        /// <para>
        /// Decrements the value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Decrement(ulong value) => --value;

        /// <summary>
        /// <para>
        /// Adds the first.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Add(ulong first, ulong second) => first + second;

        /// <summary>
        /// <para>
        /// Subtracts the first.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
349         protected override ulong Subtract(ulong first, ulong second) => first - second;
350
351         /// <summary>
352         /// <para>
353         /// Gets the header reference.
354         /// </para>
355         /// <para></para>
356         /// </summary>
357         /// <returns>
358         /// <para>A ref links header of t link</para>
359         /// <para></para>
360         /// </returns>
361         [MethodImpl(MethodImplOptions.AggressiveInlining)]
362         protected override ref LinksHeader<TLinkAddress> GetHeaderReference() => ref *Header;
363
364         /// <summary>
365         /// <para>
366         /// Gets the link data part reference using the specified link.
367         /// </para>
368         /// <para></para>
369         /// </summary>
370         /// <param name="link">
371         /// <para>The link.</para>
372         /// <para></para>
373         /// </param>
374         /// <returns>
375         /// <para>A ref raw link data part of t link</para>
376         /// <para></para>
377         /// </returns>
378         [MethodImpl(MethodImplOptions.AggressiveInlining)]
379         protected override ref RawLinkDataPart<TLinkAddress>
             ↪ GetLinkDataPartReference(TLinkAddress link) => ref LinksDataParts[link];
380
381         /// <summary>
382         /// <para>
383         /// Gets the link index part reference using the specified link.
384         /// </para>
385         /// <para></para>
386         /// </summary>
387         /// <param name="link">
388         /// <para>The link.</para>
389         /// <para></para>
390         /// </param>
391         /// <returns>
392         /// <para>A ref raw link index part of t link</para>
393         /// <para></para>
394         /// </returns>
395         [MethodImpl(MethodImplOptions.AggressiveInlining)]
396         protected override ref RawLinkIndexPart<TLinkAddress>
             ↪ GetLinkIndexPartReference(TLinkAddress link) => ref LinksIndexParts[link];
397
398         /// <summary>
399         /// <para>
400         /// Determines whether this instance first is to the left of second.
401         /// </para>
402         /// <para></para>
403         /// </summary>
404         /// <param name="first">
405         /// <para>The first.</para>
406         /// <para></para>
407         /// </param>
408         /// <param name="second">
409         /// <para>The second.</para>
410         /// <para></para>
411         /// </param>
412         /// <returns>
413         /// <para>The bool</para>
414         /// <para></para>
415         /// </returns>
416         [MethodImpl(MethodImplOptions.AggressiveInlining)]
417         protected override bool FirstIsToTheLeftOfSecond(TLinkAddress first, TLinkAddress second)
418         {
419             ref var firstLink = ref LinksDataParts[first];
420             ref var secondLink = ref LinksDataParts[second];
421             return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
                  ↪ secondLink.Source, secondLink.Target);
422         }
423
```

```
424             /// <summary>
425             /// <para>
426             /// Determines whether this instance first is to the right of second.
427             /// </para>
428             /// <para></para>
429             /// </summary>
430             /// <param name="first">
431             /// <para>The first.</para>
432             /// <para></para>
433             /// </param>
434             /// <param name="second">
435             /// <para>The second.</para>
436             /// <para></para>
437             /// </param>
438             /// <returns>
439             /// <para>The bool</para>
440             /// <para></para>
441             /// </returns>
442             [MethodImpl(MethodImplOptions.AggressiveInlining)]
443             protected override bool FirstIsToTheRightOfSecond(TLinkAddress first, TLinkAddress
                ↪ second)
444             {
445                 ref var firstLink = ref LinksDataParts[first];
446                 ref var secondLink = ref LinksDataParts[second];
447                 return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
                    ↪ secondLink.Source, secondLink.Target);
448             }
449         }
450     }
```

1.67 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksSourcesRecursionlessSizeBalanc

```
1   using System.Runtime.CompilerServices;
2   using TLinkAddress = System.UInt64;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Memory.Split.Specific
7   {
8       /// <summary>
9       /// <para>
10      /// Represents the int 64 external links sources recursionless size balanced tree methods.
11      /// </para>
12      /// <para></para>
13      /// </summary>
14      /// <seealso cref="UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase"/>
15      public unsafe class UInt64ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods :
        ↪ UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase
16      {
17          /// <summary>
18          /// <para>
19          /// Initializes a new <see
            ↪ cref="UInt64ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods"/> instance.
20          /// </para>
21          /// <para></para>
22          /// </summary>
23          /// <param name="constants">
24          /// <para>A constants.</para>
25          /// <para></para>
26          /// </param>
27          /// <param name="linksDataParts">
28          /// <para>A links data parts.</para>
29          /// <para></para>
30          /// </param>
31          /// <param name="linksIndexParts">
32          /// <para>A links index parts.</para>
33          /// <para></para>
34          /// </param>
35          /// <param name="header">
36          /// <para>A header.</para>
37          /// <para></para>
38          /// </param>
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          public UInt64ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<TLi
            ↪ nkAddress> constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
            ↪ RawLinkIndexPart<TLinkAddress>* linksIndexParts, LinksHeader<TLinkAddress>* header)
            ↪ : base(constants, linksDataParts, linksIndexParts, header) { }
41
42          /// <summary>
```

```
43          /// <para>
44          /// Gets the left reference using the specified node.
45          /// </para>
46          /// <para></para>
47          /// </summary>
48          /// <param name="node">
49          /// <para>The node.</para>
50          /// <para></para>
51          /// </param>
52          /// <returns>
53          /// <para>The ref link</para>
54          /// <para></para>
55          /// </returns>
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
            ↪ LinksIndexParts[node].LeftAsSource;
58
59          /// <summary>
60          /// <para>
61          /// Gets the right reference using the specified node.
62          /// </para>
63          /// <para></para>
64          /// </summary>
65          /// <param name="node">
66          /// <para>The node.</para>
67          /// <para></para>
68          /// </param>
69          /// <returns>
70          /// <para>The ref link</para>
71          /// <para></para>
72          /// </returns>
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
            ↪ LinksIndexParts[node].RightAsSource;
75
76          /// <summary>
77          /// <para>
78          /// Gets the left using the specified node.
79          /// </para>
80          /// <para></para>
81          /// </summary>
82          /// <param name="node">
83          /// <para>The node.</para>
84          /// <para></para>
85          /// </param>
86          /// <returns>
87          /// <para>The link</para>
88          /// <para></para>
89          /// </returns>
90          [MethodImpl(MethodImplOptions.AggressiveInlining)]
91          protected override TLinkAddress GetLeft(TLinkAddress node) =>
            ↪ LinksIndexParts[node].LeftAsSource;
92
93          /// <summary>
94          /// <para>
95          /// Gets the right using the specified node.
96          /// </para>
97          /// <para></para>
98          /// </summary>
99          /// <param name="node">
100         /// <para>The node.</para>
101         /// <para></para>
102         /// </param>
103         /// <returns>
104         /// <para>The link</para>
105         /// <para></para>
106         /// </returns>
107         [MethodImpl(MethodImplOptions.AggressiveInlining)]
108         protected override TLinkAddress GetRight(TLinkAddress node) =>
            ↪ LinksIndexParts[node].RightAsSource;
109
110         /// <summary>
111         /// <para>
112         /// Sets the left using the specified node.
113         /// </para>
114         /// <para></para>
115         /// </summary>
116         /// <param name="node">
```

```
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="left">
        /// <para>The left.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
        ↪ LinksIndexParts[node].LeftAsSource = left;

        /// <summary>
        /// <para>
        /// Sets the right using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="right">
        /// <para>The right.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
        ↪ LinksIndexParts[node].RightAsSource = right;

        /// <summary>
        /// <para>
        /// Gets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetSize(TLinkAddress node) =>
        ↪ LinksIndexParts[node].SizeAsSource;

        /// <summary>
        /// <para>
        /// Sets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="size">
        /// <para>The size.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
        ↪ LinksIndexParts[node].SizeAsSource = size;

        /// <summary>
        /// <para>
        /// Gets the tree root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetTreeRoot() => Header->RootAsSource;

```

```
        /// <summary>
        /// <para>
        /// Gets the base part value using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetBasePartValue(TLinkAddress node) =>
        ↪  LinksDataParts[node].Source;

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="firstSource">
        /// <para>The first source.</para>
        /// <para></para>
        /// </param>
        /// <param name="firstTarget">
        /// <para>The first target.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondSource">
        /// <para>The second source.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondTarget">
        /// <para>The second target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLinkAddress firstSource, TLinkAddress
        ↪  firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget)
            => firstSource < secondSource || firstSource == secondSource && firstTarget <
            ↪  secondTarget;

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the right of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="firstSource">
        /// <para>The first source.</para>
        /// <para></para>
        /// </param>
        /// <param name="firstTarget">
        /// <para>The first target.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondSource">
        /// <para>The second source.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondTarget">
        /// <para>The second target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
265         protected override bool FirstIsToTheRightOfSecond(TLinkAddress firstSource, TLinkAddress
            ↪ firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget)
266             => firstSource > secondSource || firstSource == secondSource && firstTarget >
                ↪ secondTarget;
267 
268         /// <summary>
269         /// <para>
270         /// Clears the node using the specified node.
271         /// </para>
272         /// <para></para>
273         /// </summary>
274         /// <param name="node">
275         /// <para>The node.</para>
276         /// <para></para>
277         /// </param>
278         [MethodImpl(MethodImplOptions.AggressiveInlining)]
279         protected override void ClearNode(TLinkAddress node)
280         {
281             ref var link = ref LinksIndexParts[node];
282             link.LeftAsSource = Zero;
283             link.RightAsSource = Zero;
284             link.SizeAsSource = Zero;
285         }
286     }
287 }
```

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

```
1  using System.Runtime.CompilerServices;
2  using TLinkAddress = System.UInt64;
3 
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5 
6  namespace Platform.Data.Doublets.Memory.Split.Specific
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the int 64 external links sources size balanced tree methods.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     /// <seealso cref="UInt64ExternalLinksSizeBalancedTreeMethodsBase"/>
15     public unsafe class UInt64ExternalLinksSourcesSizeBalancedTreeMethods :
       ↪ UInt64ExternalLinksSizeBalancedTreeMethodsBase
16     {
17         /// <summary>
18         /// <para>
19         /// Initializes a new <see cref="UInt64ExternalLinksSourcesSizeBalancedTreeMethods"/>
           ↪ instance.
20         /// </para>
21         /// <para></para>
22         /// </summary>
23         /// <param name="constants">
24         /// <para>A constants.</para>
25         /// <para></para>
26         /// </param>
27         /// <param name="linksDataParts">
28         /// <para>A links data parts.</para>
29         /// <para></para>
30         /// </param>
31         /// <param name="linksIndexParts">
32         /// <para>A links index parts.</para>
33         /// <para></para>
34         /// </param>
35         /// <param name="header">
36         /// <para>A header.</para>
37         /// <para></para>
38         /// </param>
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         public UInt64ExternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLinkAddress>
           ↪ constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
           ↪ RawLinkIndexPart<TLinkAddress>* linksIndexParts, LinksHeader<TLinkAddress>* header)
           ↪ : base(constants, linksDataParts, linksIndexParts, header) { }
41 
42         /// <summary>
43         /// <para>
44         /// Gets the left reference using the specified node.
45         /// </para>
46         /// <para></para>
47         /// </summary>
```

```
48          /// <param name="node">
49          /// <para>The node.</para>
50          /// <para></para>
51          /// </param>
52          /// <returns>
53          /// <para>The ref link</para>
54          /// <para></para>
55          /// </returns>
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
            ↪ LinksIndexParts[node].LeftAsSource;
58
59          /// <summary>
60          /// <para>
61          /// Gets the right reference using the specified node.
62          /// </para>
63          /// <para></para>
64          /// </summary>
65          /// <param name="node">
66          /// <para>The node.</para>
67          /// <para></para>
68          /// </param>
69          /// <returns>
70          /// <para>The ref link</para>
71          /// <para></para>
72          /// </returns>
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
            ↪ LinksIndexParts[node].RightAsSource;
75
76          /// <summary>
77          /// <para>
78          /// Gets the left using the specified node.
79          /// </para>
80          /// <para></para>
81          /// </summary>
82          /// <param name="node">
83          /// <para>The node.</para>
84          /// <para></para>
85          /// </param>
86          /// <returns>
87          /// <para>The link</para>
88          /// <para></para>
89          /// </returns>
90          [MethodImpl(MethodImplOptions.AggressiveInlining)]
91          protected override TLinkAddress GetLeft(TLinkAddress node) =>
            ↪ LinksIndexParts[node].LeftAsSource;
92
93          /// <summary>
94          /// <para>
95          /// Gets the right using the specified node.
96          /// </para>
97          /// <para></para>
98          /// </summary>
99          /// <param name="node">
100         /// <para>The node.</para>
101         /// <para></para>
102         /// </param>
103         /// <returns>
104         /// <para>The link</para>
105         /// <para></para>
106         /// </returns>
107         [MethodImpl(MethodImplOptions.AggressiveInlining)]
108         protected override TLinkAddress GetRight(TLinkAddress node) =>
            ↪ LinksIndexParts[node].RightAsSource;
109
110         /// <summary>
111         /// <para>
112         /// Sets the left using the specified node.
113         /// </para>
114         /// <para></para>
115         /// </summary>
116         /// <param name="node">
117         /// <para>The node.</para>
118         /// <para></para>
119         /// </param>
120         /// <param name="left">
121         /// <para>The left.</para>
```

```
122             /// <para></para>
123             /// </param>
124             [MethodImpl(MethodImplOptions.AggressiveInlining)]
125             protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
                ↪ LinksIndexParts[node].LeftAsSource = left;
126
127             /// <summary>
128             /// <para>
129             /// Sets the right using the specified node.
130             /// </para>
131             /// <para></para>
132             /// </summary>
133             /// <param name="node">
134             /// <para>The node.</para>
135             /// <para></para>
136             /// </param>
137             /// <param name="right">
138             /// <para>The right.</para>
139             /// <para></para>
140             /// </param>
141             [MethodImpl(MethodImplOptions.AggressiveInlining)]
142             protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
                ↪ LinksIndexParts[node].RightAsSource = right;
143
144             /// <summary>
145             /// <para>
146             /// Gets the size using the specified node.
147             /// </para>
148             /// <para></para>
149             /// </summary>
150             /// <param name="node">
151             /// <para>The node.</para>
152             /// <para></para>
153             /// </param>
154             /// <returns>
155             /// <para>The link</para>
156             /// <para></para>
157             /// </returns>
158             [MethodImpl(MethodImplOptions.AggressiveInlining)]
159             protected override TLinkAddress GetSize(TLinkAddress node) =>
                ↪ LinksIndexParts[node].SizeAsSource;
160
161             /// <summary>
162             /// <para>
163             /// Sets the size using the specified node.
164             /// </para>
165             /// <para></para>
166             /// </summary>
167             /// <param name="node">
168             /// <para>The node.</para>
169             /// <para></para>
170             /// </param>
171             /// <param name="size">
172             /// <para>The size.</para>
173             /// <para></para>
174             /// </param>
175             [MethodImpl(MethodImplOptions.AggressiveInlining)]
176             protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
                ↪ LinksIndexParts[node].SizeAsSource = size;
177
178             /// <summary>
179             /// <para>
180             /// Gets the tree root.
181             /// </para>
182             /// <para></para>
183             /// </summary>
184             /// <returns>
185             /// <para>The link</para>
186             /// <para></para>
187             /// </returns>
188             [MethodImpl(MethodImplOptions.AggressiveInlining)]
189             protected override TLinkAddress GetTreeRoot() => Header->RootAsSource;
190
191             /// <summary>
192             /// <para>
193             /// Gets the base part value using the specified node.
194             /// </para>
195             /// <para></para>
```

```
196             /// </summary>
197             /// <param name="node">
198             /// <para>The node.</para>
199             /// <para></para>
200             /// </param>
201             /// <returns>
202             /// <para>The link</para>
203             /// <para></para>
204             /// </returns>
205             [MethodImpl(MethodImplOptions.AggressiveInlining)]
206             protected override TLinkAddress GetBasePartValue(TLinkAddress node) =>
                ↪ LinksDataParts[node].Source;
207
208             /// <summary>
209             /// <para>
210             /// Determines whether this instance first is to the left of second.
211             /// </para>
212             /// <para></para>
213             /// </summary>
214             /// <param name="firstSource">
215             /// <para>The first source.</para>
216             /// <para></para>
217             /// </param>
218             /// <param name="firstTarget">
219             /// <para>The first target.</para>
220             /// <para></para>
221             /// </param>
222             /// <param name="secondSource">
223             /// <para>The second source.</para>
224             /// <para></para>
225             /// </param>
226             /// <param name="secondTarget">
227             /// <para>The second target.</para>
228             /// <para></para>
229             /// </param>
230             /// <returns>
231             /// <para>The bool</para>
232             /// <para></para>
233             /// </returns>
234             [MethodImpl(MethodImplOptions.AggressiveInlining)]
235             protected override bool FirstIsToTheLeftOfSecond(TLinkAddress firstSource, TLinkAddress
                ↪ firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget)
                => firstSource < secondSource || firstSource == secondSource && firstTarget <
236                 ↪ secondTarget;
237
238             /// <summary>
239             /// <para>
240             /// Determines whether this instance first is to the right of second.
241             /// </para>
242             /// <para></para>
243             /// </summary>
244             /// <param name="firstSource">
245             /// <para>The first source.</para>
246             /// <para></para>
247             /// </param>
248             /// <param name="firstTarget">
249             /// <para>The first target.</para>
250             /// <para></para>
251             /// </param>
252             /// <param name="secondSource">
253             /// <para>The second source.</para>
254             /// <para></para>
255             /// </param>
256             /// <param name="secondTarget">
257             /// <para>The second target.</para>
258             /// <para></para>
259             /// </param>
260             /// <returns>
261             /// <para>The bool</para>
262             /// <para></para>
263             /// </returns>
264             [MethodImpl(MethodImplOptions.AggressiveInlining)]
265             protected override bool FirstIsToTheRightOfSecond(TLinkAddress firstSource, TLinkAddress
                ↪ firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget)
                => firstSource > secondSource || firstSource == secondSource && firstTarget >
266                 ↪ secondTarget;
267
```

```
        /// <summary>
        /// <para>
        /// Clears the node using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLinkAddress node)
        {
            ref var link = ref LinksIndexParts[node];
            link.LeftAsSource = Zero;
            link.RightAsSource = Zero;
            link.SizeAsSource = Zero;
        }
    }
}
```

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

```
using System.Runtime.CompilerServices;
using TLinkAddress = System.UInt64;

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    /// <summary>
    /// <para>
    /// Represents the int 64 external links targets recursionless size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase"/>
    public unsafe class UInt64ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods :
    ↪ UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see
        ↪ cref="UInt64ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksDataParts">
        /// <para>A links data parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksIndexParts">
        /// <para>A links index parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<TLi
        ↪ nkAddress> constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
        ↪ RawLinkIndexPart<TLinkAddress>* linksIndexParts, LinksHeader<TLinkAddress>* header)
        ↪ : base(constants, linksDataParts, linksIndexParts, header) { }

        /// <summary>
        /// <para>
        /// Gets the left reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
```

```
53          /// <para>The ref link</para>
54          /// <para></para>
55          /// </returns>
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
            ↪ LinksIndexParts[node].LeftAsTarget;
58
59          /// <summary>
60          /// <para>
61          /// Gets the right reference using the specified node.
62          /// </para>
63          /// <para></para>
64          /// </summary>
65          /// <param name="node">
66          /// <para>The node.</para>
67          /// <para></para>
68          /// </param>
69          /// <returns>
70          /// <para>The ref link</para>
71          /// <para></para>
72          /// </returns>
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
            ↪ LinksIndexParts[node].RightAsTarget;
75
76          /// <summary>
77          /// <para>
78          /// Gets the left using the specified node.
79          /// </para>
80          /// <para></para>
81          /// </summary>
82          /// <param name="node">
83          /// <para>The node.</para>
84          /// <para></para>
85          /// </param>
86          /// <returns>
87          /// <para>The link</para>
88          /// <para></para>
89          /// </returns>
90          [MethodImpl(MethodImplOptions.AggressiveInlining)]
91          protected override TLinkAddress GetLeft(TLinkAddress node) =>
            ↪ LinksIndexParts[node].LeftAsTarget;
92
93          /// <summary>
94          /// <para>
95          /// Gets the right using the specified node.
96          /// </para>
97          /// <para></para>
98          /// </summary>
99          /// <param name="node">
100         /// <para>The node.</para>
101         /// <para></para>
102         /// </param>
103         /// <returns>
104         /// <para>The link</para>
105         /// <para></para>
106         /// </returns>
107         [MethodImpl(MethodImplOptions.AggressiveInlining)]
108         protected override TLinkAddress GetRight(TLinkAddress node) =>
            ↪ LinksIndexParts[node].RightAsTarget;
109
110         /// <summary>
111         /// <para>
112         /// Sets the left using the specified node.
113         /// </para>
114         /// <para></para>
115         /// </summary>
116         /// <param name="node">
117         /// <para>The node.</para>
118         /// <para></para>
119         /// </param>
120         /// <param name="left">
121         /// <para>The left.</para>
122         /// <para></para>
123         /// </param>
124         [MethodImpl(MethodImplOptions.AggressiveInlining)]
125         protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
            ↪ LinksIndexParts[node].LeftAsTarget = left;
```

```
126
127         /// <summary>
128         /// <para>
129         /// Sets the right using the specified node.
130         /// </para>
131         /// <para></para>
132         /// </summary>
133         /// <param name="node">
134         /// <para>The node.</para>
135         /// <para></para>
136         /// </param>
137         /// <param name="right">
138         /// <para>The right.</para>
139         /// <para></para>
140         /// </param>
141         [MethodImpl(MethodImplOptions.AggressiveInlining)]
142         protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
            ↪ LinksIndexParts[node].RightAsTarget = right;
143
144         /// <summary>
145         /// <para>
146         /// Gets the size using the specified node.
147         /// </para>
148         /// <para></para>
149         /// </summary>
150         /// <param name="node">
151         /// <para>The node.</para>
152         /// <para></para>
153         /// </param>
154         /// <returns>
155         /// <para>The link</para>
156         /// <para></para>
157         /// </returns>
158         [MethodImpl(MethodImplOptions.AggressiveInlining)]
159         protected override TLinkAddress GetSize(TLinkAddress node) =>
            ↪ LinksIndexParts[node].SizeAsTarget;
160
161         /// <summary>
162         /// <para>
163         /// Sets the size using the specified node.
164         /// </para>
165         /// <para></para>
166         /// </summary>
167         /// <param name="node">
168         /// <para>The node.</para>
169         /// <para></para>
170         /// </param>
171         /// <param name="size">
172         /// <para>The size.</para>
173         /// <para></para>
174         /// </param>
175         [MethodImpl(MethodImplOptions.AggressiveInlining)]
176         protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
            ↪ LinksIndexParts[node].SizeAsTarget = size;
177
178         /// <summary>
179         /// <para>
180         /// Gets the tree root.
181         /// </para>
182         /// <para></para>
183         /// </summary>
184         /// <returns>
185         /// <para>The link</para>
186         /// <para></para>
187         /// </returns>
188         [MethodImpl(MethodImplOptions.AggressiveInlining)]
189         protected override TLinkAddress GetTreeRoot() => Header->RootAsTarget;
190
191         /// <summary>
192         /// <para>
193         /// Gets the base part value using the specified node.
194         /// </para>
195         /// <para></para>
196         /// </summary>
197         /// <param name="node">
198         /// <para>The node.</para>
199         /// <para></para>
200         /// </param>
```

```
201         /// <returns>
202         /// <para>The link</para>
203         /// <para></para>
204         /// </returns>
205         [MethodImpl(MethodImplOptions.AggressiveInlining)]
206         protected override TLinkAddress GetBasePartValue(TLinkAddress node) =>
            ↪ LinksDataParts[node].Target;
207
208         /// <summary>
209         /// <para>
210         /// Determines whether this instance first is to the left of second.
211         /// </para>
212         /// <para></para>
213         /// </summary>
214         /// <param name="firstSource">
215         /// <para>The first source.</para>
216         /// <para></para>
217         /// </param>
218         /// <param name="firstTarget">
219         /// <para>The first target.</para>
220         /// <para></para>
221         /// </param>
222         /// <param name="secondSource">
223         /// <para>The second source.</para>
224         /// <para></para>
225         /// </param>
226         /// <param name="secondTarget">
227         /// <para>The second target.</para>
228         /// <para></para>
229         /// </param>
230         /// <returns>
231         /// <para>The bool</para>
232         /// <para></para>
233         /// </returns>
234         [MethodImpl(MethodImplOptions.AggressiveInlining)]
235         protected override bool FirstIsToTheLeftOfSecond(TLinkAddress firstSource, TLinkAddress
            ↪ firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget)
                => firstTarget < secondTarget || firstTarget == secondTarget && firstSource <
236                 ↪ secondSource;
237
238         /// <summary>
239         /// <para>
240         /// Determines whether this instance first is to the right of second.
241         /// </para>
242         /// <para></para>
243         /// </summary>
244         /// <param name="firstSource">
245         /// <para>The first source.</para>
246         /// <para></para>
247         /// </param>
248         /// <param name="firstTarget">
249         /// <para>The first target.</para>
250         /// <para></para>
251         /// </param>
252         /// <param name="secondSource">
253         /// <para>The second source.</para>
254         /// <para></para>
255         /// </param>
256         /// <param name="secondTarget">
257         /// <para>The second target.</para>
258         /// <para></para>
259         /// </param>
260         /// <returns>
261         /// <para>The bool</para>
262         /// <para></para>
263         /// </returns>
264         [MethodImpl(MethodImplOptions.AggressiveInlining)]
265         protected override bool FirstIsToTheRightOfSecond(TLinkAddress firstSource, TLinkAddress
            ↪ firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget)
                => firstTarget > secondTarget || firstTarget == secondTarget && firstSource >
266                 ↪ secondSource;
267
268         /// <summary>
269         /// <para>
270         /// Clears the node using the specified node.
271         /// </para>
272         /// <para></para>
```

```
273         /// </summary>
274         /// <param name="node">
275         /// <para>The node.</para>
276         /// <para></para>
277         /// </param>
278         [MethodImpl(MethodImplOptions.AggressiveInlining)]
279         protected override void ClearNode(TLinkAddress node)
280         {
281             ref var link = ref LinksIndexParts[node];
282             link.LeftAsTarget = Zero;
283             link.RightAsTarget = Zero;
284             link.SizeAsTarget = Zero;
285         }
286     }
287 }
```

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

```
1  using System.Runtime.CompilerServices;
2  using TLinkAddress = System.UInt64;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.Split.Specific
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the int 64 external links targets size balanced tree methods.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     /// <seealso cref="UInt64ExternalLinksSizeBalancedTreeMethodsBase"/>
15     public unsafe class UInt64ExternalLinksTargetsSizeBalancedTreeMethods :
       ↪  UInt64ExternalLinksSizeBalancedTreeMethodsBase
16     {
17         /// <summary>
18         /// <para>
19         /// Initializes a new <see cref="UInt64ExternalLinksTargetsSizeBalancedTreeMethods"/>
           ↪  instance.
20         /// </para>
21         /// <para></para>
22         /// </summary>
23         /// <param name="constants">
24         /// <para>A constants.</para>
25         /// <para></para>
26         /// </param>
27         /// <param name="linksDataParts">
28         /// <para>A links data parts.</para>
29         /// <para></para>
30         /// </param>
31         /// <param name="linksIndexParts">
32         /// <para>A links index parts.</para>
33         /// <para></para>
34         /// </param>
35         /// <param name="header">
36         /// <para>A header.</para>
37         /// <para></para>
38         /// </param>
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         public UInt64ExternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLinkAddress>
           ↪  constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
           ↪  RawLinkIndexPart<TLinkAddress>* linksIndexParts, LinksHeader<TLinkAddress>* header)
           ↪  : base(constants, linksDataParts, linksIndexParts, header) { }
41
42         /// <summary>
43         /// <para>
44         /// Gets the left reference using the specified node.
45         /// </para>
46         /// <para></para>
47         /// </summary>
48         /// <param name="node">
49         /// <para>The node.</para>
50         /// <para></para>
51         /// </param>
52         /// <returns>
53         /// <para>The ref link</para>
54         /// <para></para>
55         /// </returns>
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
57          protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
            ↪ LinksIndexParts[node].LeftAsTarget;
58
59          /// <summary>
60          /// <para>
61          /// Gets the right reference using the specified node.
62          /// </para>
63          /// <para></para>
64          /// </summary>
65          /// <param name="node">
66          /// <para>The node.</para>
67          /// <para></para>
68          /// </param>
69          /// <returns>
70          /// <para>The ref link</para>
71          /// <para></para>
72          /// </returns>
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
            ↪ LinksIndexParts[node].RightAsTarget;
75
76          /// <summary>
77          /// <para>
78          /// Gets the left using the specified node.
79          /// </para>
80          /// <para></para>
81          /// </summary>
82          /// <param name="node">
83          /// <para>The node.</para>
84          /// <para></para>
85          /// </param>
86          /// <returns>
87          /// <para>The link</para>
88          /// <para></para>
89          /// </returns>
90          [MethodImpl(MethodImplOptions.AggressiveInlining)]
91          protected override TLinkAddress GetLeft(TLinkAddress node) =>
            ↪ LinksIndexParts[node].LeftAsTarget;
92
93          /// <summary>
94          /// <para>
95          /// Gets the right using the specified node.
96          /// </para>
97          /// <para></para>
98          /// </summary>
99          /// <param name="node">
100         /// <para>The node.</para>
101         /// <para></para>
102         /// </param>
103         /// <returns>
104         /// <para>The link</para>
105         /// <para></para>
106         /// </returns>
107         [MethodImpl(MethodImplOptions.AggressiveInlining)]
108         protected override TLinkAddress GetRight(TLinkAddress node) =>
            ↪ LinksIndexParts[node].RightAsTarget;
109
110         /// <summary>
111         /// <para>
112         /// Sets the left using the specified node.
113         /// </para>
114         /// <para></para>
115         /// </summary>
116         /// <param name="node">
117         /// <para>The node.</para>
118         /// <para></para>
119         /// </param>
120         /// <param name="left">
121         /// <para>The left.</para>
122         /// <para></para>
123         /// </param>
124         [MethodImpl(MethodImplOptions.AggressiveInlining)]
125         protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
            ↪ LinksIndexParts[node].LeftAsTarget = left;
126
127         /// <summary>
128         /// <para>
```

```
129             /// Sets the right using the specified node.
130             /// </para>
131             /// <para></para>
132             /// </summary>
133             /// <param name="node">
134             /// <para>The node.</para>
135             /// <para></para>
136             /// </param>
137             /// <param name="right">
138             /// <para>The right.</para>
139             /// <para></para>
140             /// </param>
141             [MethodImpl(MethodImplOptions.AggressiveInlining)]
142             protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
                ↪ LinksIndexParts[node].RightAsTarget = right;
143
144             /// <summary>
145             /// <para>
146             /// Gets the size using the specified node.
147             /// </para>
148             /// <para></para>
149             /// </summary>
150             /// <param name="node">
151             /// <para>The node.</para>
152             /// <para></para>
153             /// </param>
154             /// <returns>
155             /// <para>The link</para>
156             /// <para></para>
157             /// </returns>
158             [MethodImpl(MethodImplOptions.AggressiveInlining)]
159             protected override TLinkAddress GetSize(TLinkAddress node) =>
                ↪ LinksIndexParts[node].SizeAsTarget;
160
161             /// <summary>
162             /// <para>
163             /// Sets the size using the specified node.
164             /// </para>
165             /// <para></para>
166             /// </summary>
167             /// <param name="node">
168             /// <para>The node.</para>
169             /// <para></para>
170             /// </param>
171             /// <param name="size">
172             /// <para>The size.</para>
173             /// <para></para>
174             /// </param>
175             [MethodImpl(MethodImplOptions.AggressiveInlining)]
176             protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
                ↪ LinksIndexParts[node].SizeAsTarget = size;
177
178             /// <summary>
179             /// <para>
180             /// Gets the tree root.
181             /// </para>
182             /// <para></para>
183             /// </summary>
184             /// <returns>
185             /// <para>The link</para>
186             /// <para></para>
187             /// </returns>
188             [MethodImpl(MethodImplOptions.AggressiveInlining)]
189             protected override TLinkAddress GetTreeRoot() => Header->RootAsTarget;
190
191             /// <summary>
192             /// <para>
193             /// Gets the base part value using the specified node.
194             /// </para>
195             /// <para></para>
196             /// </summary>
197             /// <param name="node">
198             /// <para>The node.</para>
199             /// <para></para>
200             /// </param>
201             /// <returns>
202             /// <para>The link</para>
203             /// <para></para>
```

```
204         /// </returns>
205         [MethodImpl(MethodImplOptions.AggressiveInlining)]
206         protected override TLinkAddress GetBasePartValue(TLinkAddress node) =>
            ↪ LinksDataParts[node].Target;
207
208         /// <summary>
209         /// <para>
210         /// Determines whether this instance first is to the left of second.
211         /// </para>
212         /// <para></para>
213         /// </summary>
214         /// <param name="firstSource">
215         /// <para>The first source.</para>
216         /// <para></para>
217         /// </param>
218         /// <param name="firstTarget">
219         /// <para>The first target.</para>
220         /// <para></para>
221         /// </param>
222         /// <param name="secondSource">
223         /// <para>The second source.</para>
224         /// <para></para>
225         /// </param>
226         /// <param name="secondTarget">
227         /// <para>The second target.</para>
228         /// <para></para>
229         /// </param>
230         /// <returns>
231         /// <para>The bool</para>
232         /// <para></para>
233         /// </returns>
234         [MethodImpl(MethodImplOptions.AggressiveInlining)]
235         protected override bool FirstIsToTheLeftOfSecond(TLinkAddress firstSource, TLinkAddress
            ↪ firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget)
236             => firstTarget < secondTarget || firstTarget == secondTarget && firstSource <
                ↪ secondSource;
237
238         /// <summary>
239         /// <para>
240         /// Determines whether this instance first is to the right of second.
241         /// </para>
242         /// <para></para>
243         /// </summary>
244         /// <param name="firstSource">
245         /// <para>The first source.</para>
246         /// <para></para>
247         /// </param>
248         /// <param name="firstTarget">
249         /// <para>The first target.</para>
250         /// <para></para>
251         /// </param>
252         /// <param name="secondSource">
253         /// <para>The second source.</para>
254         /// <para></para>
255         /// </param>
256         /// <param name="secondTarget">
257         /// <para>The second target.</para>
258         /// <para></para>
259         /// </param>
260         /// <returns>
261         /// <para>The bool</para>
262         /// <para></para>
263         /// </returns>
264         [MethodImpl(MethodImplOptions.AggressiveInlining)]
265         protected override bool FirstIsToTheRightOfSecond(TLinkAddress firstSource, TLinkAddress
            ↪ firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget)
266             => firstTarget > secondTarget || firstTarget == secondTarget && firstSource >
                ↪ secondSource;
267
268         /// <summary>
269         /// <para>
270         /// Clears the node using the specified node.
271         /// </para>
272         /// <para></para>
273         /// </summary>
274         /// <param name="node">
275         /// <para>The node.</para>
```

```
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLinkAddress node)
        {
            ref var link = ref LinksIndexParts[node];
            link.LeftAsTarget = Zero;
            link.RightAsTarget = Zero;
            link.SizeAsTarget = Zero;
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    /// <summary>
    /// <para>
    /// Represents the int 64 internal links recursionless size balanced tree methods base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="InternalLinksRecursionlessSizeBalancedTreeMethodsBase{TLinkAddress}"/>
    public unsafe abstract class UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase :
    ↪  InternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// The links data parts.
        /// </para>
        /// <para></para>
        /// </summary>
        protected new readonly RawLinkDataPart<TLinkAddress>* LinksDataParts;
        /// <summary>
        /// <para>
        /// The links index parts.
        /// </para>
        /// <para></para>
        /// </summary>
        protected new readonly RawLinkIndexPart<TLinkAddress>* LinksIndexParts;
        /// <summary>
        /// <para>
        /// The header.
        /// </para>
        /// <para></para>
        /// </summary>
        protected new readonly LinksHeader<TLinkAddress>* Header;

        /// <summary>
        /// <para>
        /// Initializes a new <see
        ↪  cref="UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksDataParts">
        /// <para>A links data parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksIndexParts">
        /// <para>A links index parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
63          protected UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLi
            ↪  nkAddress> constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
            ↪  RawLinkIndexPart<TLinkAddress>* linksIndexParts, LinksHeader<TLinkAddress>* header)
64              : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
65          {
66              LinksDataParts = linksDataParts;
67              LinksIndexParts = linksIndexParts;
68              Header = header;
69          }
70
71          /// <summary>
72          /// <para>
73          /// Gets the zero.
74          /// </para>
75          /// <para></para>
76          /// </summary>
77          /// <returns>
78          /// <para>The ulong</para>
79          /// <para></para>
80          /// </returns>
81          [MethodImpl(MethodImplOptions.AggressiveInlining)]
82          protected override ulong GetZero() => 0UL;
83
84          /// <summary>
85          /// <para>
86          /// Determines whether this instance equal to zero.
87          /// </para>
88          /// <para></para>
89          /// </summary>
90          /// <param name="value">
91          /// <para>The value.</para>
92          /// <para></para>
93          /// </param>
94          /// <returns>
95          /// <para>The bool</para>
96          /// <para></para>
97          /// </returns>
98          [MethodImpl(MethodImplOptions.AggressiveInlining)]
99          protected override bool EqualToZero(ulong value) => value == 0UL;
100
101         /// <summary>
102         /// <para>
103         /// Determines whether this instance are equal.
104         /// </para>
105         /// <para></para>
106         /// </summary>
107         /// <param name="first">
108         /// <para>The first.</para>
109         /// <para></para>
110         /// </param>
111         /// <param name="second">
112         /// <para>The second.</para>
113         /// <para></para>
114         /// </param>
115         /// <returns>
116         /// <para>The bool</para>
117         /// <para></para>
118         /// </returns>
119         [MethodImpl(MethodImplOptions.AggressiveInlining)]
120         protected override bool AreEqual(ulong first, ulong second) => first == second;
121
122         /// <summary>
123         /// <para>
124         /// Determines whether this instance greater than zero.
125         /// </para>
126         /// <para></para>
127         /// </summary>
128         /// <param name="value">
129         /// <para>The value.</para>
130         /// <para></para>
131         /// </param>
132         /// <returns>
133         /// <para>The bool</para>
134         /// <para></para>
135         /// </returns>
136         [MethodImpl(MethodImplOptions.AggressiveInlining)]
137         protected override bool GreaterThanZero(ulong value) => value > 0UL;
138
```

```
        /// <summary>
        /// <para>
        /// Determines whether this instance greater than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThan(ulong first, ulong second) => first > second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater or equal than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater or equal than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
        ↪  always true for ulong

        /// <summary>
        /// <para>
        /// Determines whether this instance less or equal than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
        ↪  always >= 0 for ulong

```

```
        /// <summary>
        /// <para>
        /// Determines whether this instance less or equal than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;

        /// <summary>
        /// <para>
        /// Determines whether this instance less than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
        ↪  for ulong

        /// <summary>
        /// <para>
        /// Determines whether this instance less than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessThan(ulong first, ulong second) => first < second;

        /// <summary>
        /// <para>
        /// Increments the value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Increment(ulong value) => ++value;

        /// <summary>
```

```
        /// <para>
        /// Decrements the value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Decrement(ulong value) => --value;

        /// <summary>
        /// <para>
        /// Adds the first.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Add(ulong first, ulong second) => first + second;

        /// <summary>
        /// <para>
        /// Subtracts the first.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Subtract(ulong first, ulong second) => first - second;

        /// <summary>
        /// <para>
        /// Gets the link data part reference using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link data part of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkDataPart<TLinkAddress>
        ↪  GetLinkDataPartReference(TLinkAddress link) => ref LinksDataParts[link];

        /// <summary>
        /// <para>
```

```
        /// Gets the link index part reference using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link index part of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkIndexPart<TLinkAddress>
        ↪  GetLinkIndexPartReference(TLinkAddress link) => ref LinksIndexParts[link];

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLinkAddress first, TLinkAddress
        ↪  second) => GetKeyPartValue(first) < GetKeyPartValue(second);

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the right of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLinkAddress first, TLinkAddress
        ↪  second) => GetKeyPartValue(first) > GetKeyPartValue(second);
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    /// <summary>
    /// <para>
    /// Represents the int 64 internal links size balanced tree methods base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="InternalLinksSizeBalancedTreeMethodsBase{TLinkAddress}"/>
```

```
16      public unsafe abstract class UInt64InternalLinksSizeBalancedTreeMethodsBase :
        ↪ InternalLinksSizeBalancedTreeMethodsBase<TLinkAddress>
17      {
18          /// <summary>
19          /// <para>
20          /// The links data parts.
21          /// </para>
22          /// <para></para>
23          /// </summary>
24          protected new readonly RawLinkDataPart<TLinkAddress>* LinksDataParts;
25          /// <summary>
26          /// <para>
27          /// The links index parts.
28          /// </para>
29          /// <para></para>
30          /// </summary>
31          protected new readonly RawLinkIndexPart<TLinkAddress>* LinksIndexParts;
32          /// <summary>
33          /// <para>
34          /// The header.
35          /// </para>
36          /// <para></para>
37          /// </summary>
38          protected new readonly LinksHeader<TLinkAddress>* Header;
39
40          /// <summary>
41          /// <para>
42          /// Initializes a new <see cref="UInt64InternalLinksSizeBalancedTreeMethodsBase"/>
            ↪ instance.
43          /// </para>
44          /// <para></para>
45          /// </summary>
46          /// <param name="constants">
47          /// <para>A constants.</para>
48          /// <para></para>
49          /// </param>
50          /// <param name="linksDataParts">
51          /// <para>A links data parts.</para>
52          /// <para></para>
53          /// </param>
54          /// <param name="linksIndexParts">
55          /// <para>A links index parts.</para>
56          /// <para></para>
57          /// </param>
58          /// <param name="header">
59          /// <para>A header.</para>
60          /// <para></para>
61          /// </param>
62          [MethodImpl(MethodImplOptions.AggressiveInlining)]
63          protected UInt64InternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLinkAddress>
            ↪ constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
            ↪ RawLinkIndexPart<TLinkAddress>* linksIndexParts, LinksHeader<TLinkAddress>* header)
64              : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
65          {
66              LinksDataParts = linksDataParts;
67              LinksIndexParts = linksIndexParts;
68              Header = header;
69          }
70
71          /// <summary>
72          /// <para>
73          /// Gets the zero.
74          /// </para>
75          /// <para></para>
76          /// </summary>
77          /// <returns>
78          /// <para>The ulong</para>
79          /// <para></para>
80          /// </returns>
81          [MethodImpl(MethodImplOptions.AggressiveInlining)]
82          protected override ulong GetZero() => 0UL;
83
84          /// <summary>
85          /// <para>
86          /// Determines whether this instance equal to zero.
87          /// </para>
88          /// <para></para>
89          /// </summary>
```

```
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool EqualToZero(ulong value) => value == 0UL;

        /// <summary>
        /// <para>
        /// Determines whether this instance are equal.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool AreEqual(ulong first, ulong second) => first == second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThanZero(ulong value) => value > 0UL;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThan(ulong first, ulong second) => first > second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater or equal than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
```

```
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater or equal than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
        ↪  always true for ulong

        /// <summary>
        /// <para>
        /// Determines whether this instance less or equal than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
        ↪  always >= 0 for ulong

        /// <summary>
        /// <para>
        /// Determines whether this instance less or equal than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;

        /// <summary>
        /// <para>
        /// Determines whether this instance less than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
```

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

        /// <summary>
        /// <para>
        /// Determines whether this instance less than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessThan(ulong first, ulong second) => first < second;

        /// <summary>
        /// <para>
        /// Increments the value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Increment(ulong value) => ++value;

        /// <summary>
        /// <para>
        /// Decrements the value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Decrement(ulong value) => --value;

        /// <summary>
        /// <para>
        /// Adds the first.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
```

```
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Add(ulong first, ulong second) => first + second;

        /// <summary>
        /// <para>
        /// Subtracts the first.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Subtract(ulong first, ulong second) => first - second;

        /// <summary>
        /// <para>
        /// Gets the link data part reference using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link data part of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkDataPart<TLinkAddress>
        ↪  GetLinkDataPartReference(TLinkAddress link) => ref LinksDataParts[link];

        /// <summary>
        /// <para>
        /// Gets the link index part reference using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link index part of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkIndexPart<TLinkAddress>
        ↪  GetLinkIndexPartReference(TLinkAddress link) => ref LinksIndexParts[link];

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
```

```
397         /// </param>
398         /// <returns>
399         /// <para>The bool</para>
400         /// <para></para>
401         /// </returns>
402         [MethodImpl(MethodImplOptions.AggressiveInlining)]
403         protected override bool FirstIsToTheLeftOfSecond(TLinkAddress first, TLinkAddress
            ↪ second) => GetKeyPartValue(first) < GetKeyPartValue(second);
404
405         /// <summary>
406         /// <para>
407         /// Determines whether this instance first is to the right of second.
408         /// </para>
409         /// <para></para>
410         /// </summary>
411         /// <param name="first">
412         /// <para>The first.</para>
413         /// <para></para>
414         /// </param>
415         /// <param name="second">
416         /// <para>The second.</para>
417         /// <para></para>
418         /// </param>
419         /// <returns>
420         /// <para>The bool</para>
421         /// <para></para>
422         /// </returns>
423         [MethodImpl(MethodImplOptions.AggressiveInlining)]
424         protected override bool FirstIsToTheRightOfSecond(TLinkAddress first, TLinkAddress
            ↪ second) => GetKeyPartValue(first) > GetKeyPartValue(second);
425     }
426 }
```

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

```
1  using System.Runtime.CompilerServices;
2  using TLinkAddress = System.UInt64;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.Split.Generic
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the int 64 internal links sources linked list methods.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     /// <seealso cref="InternalLinksSourcesLinkedListMethods{TLinkAddress}"/>
15     public unsafe class UInt64InternalLinksSourcesLinkedListMethods :
       ↪ InternalLinksSourcesLinkedListMethods<TLinkAddress>
16     {
17         private readonly RawLinkDataPart<TLinkAddress>* _linksDataParts;
18         private readonly RawLinkIndexPart<TLinkAddress>* _linksIndexParts;
19
20         /// <summary>
21         /// <para>
22         /// Initializes a new <see cref="UInt64InternalLinksSourcesLinkedListMethods"/> instance.
23         /// </para>
24         /// <para></para>
25         /// </summary>
26         /// <param name="constants">
27         /// <para>A constants.</para>
28         /// <para></para>
29         /// </param>
30         /// <param name="linksDataParts">
31         /// <para>A links data parts.</para>
32         /// <para></para>
33         /// </param>
34         /// <param name="linksIndexParts">
35         /// <para>A links index parts.</para>
36         /// <para></para>
37         /// </param>
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public UInt64InternalLinksSourcesLinkedListMethods(LinksConstants<TLinkAddress>
           ↪ constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
           ↪ RawLinkIndexPart<TLinkAddress>* linksIndexParts)
40             : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts)
41         {
```

```
            _linksDataParts = linksDataParts;
            _linksIndexParts = linksIndexParts;
        }

        /// <summary>
        /// <para>
        /// Gets the link data part reference using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link data part of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkDataPart<TLinkAddress>
        ↪ GetLinkDataPartReference(TLinkAddress link) => ref _linksDataParts[link];

        /// <summary>
        /// <para>
        /// Gets the link index part reference using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link index part of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkIndexPart<TLinkAddress>
        ↪ GetLinkIndexPartReference(TLinkAddress link) => ref _linksIndexParts[link];
    }
}
```

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

```
using System.Runtime.CompilerServices;
using TLinkAddress = System.UInt64;

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    /// <summary>
    /// <para>
    /// Represents the int 64 internal links sources recursionless size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase"/>
    public unsafe class UInt64InternalLinksSourcesRecursionlessSizeBalancedTreeMethods :
    ↪ UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see
        ↪ cref="UInt64InternalLinksSourcesRecursionlessSizeBalancedTreeMethods"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksDataParts">
        /// <para>A links data parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksIndexParts">
        /// <para>A links index parts.</para>
        /// <para></para>
```

```
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<TLi⌋
        ↪  nkAddress> constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
        ↪  RawLinkIndexPart<TLinkAddress>* linksIndexParts, LinksHeader<TLinkAddress>* header)
        ↪  : base(constants, linksDataParts, linksIndexParts, header) { }

        /// <summary>
        /// <para>
        /// Gets the left reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
        ↪  LinksIndexParts[node].LeftAsSource;

        /// <summary>
        /// <para>
        /// Gets the right reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
        ↪  LinksIndexParts[node].RightAsSource;

        /// <summary>
        /// <para>
        /// Gets the left using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetLeft(TLinkAddress node) =>
        ↪  LinksIndexParts[node].LeftAsSource;

        /// <summary>
        /// <para>
        /// Gets the right using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
```

```
106         /// </returns>
107         [MethodImpl(MethodImplOptions.AggressiveInlining)]
108         protected override TLinkAddress GetRight(TLinkAddress node) =>
            ↪ LinksIndexParts[node].RightAsSource;
109
110         /// <summary>
111         /// <para>
112         /// Sets the left using the specified node.
113         /// </para>
114         /// <para></para>
115         /// </summary>
116         /// <param name="node">
117         /// <para>The node.</para>
118         /// <para></para>
119         /// </param>
120         /// <param name="left">
121         /// <para>The left.</para>
122         /// <para></para>
123         /// </param>
124         [MethodImpl(MethodImplOptions.AggressiveInlining)]
125         protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
            ↪ LinksIndexParts[node].LeftAsSource = left;
126
127         /// <summary>
128         /// <para>
129         /// Sets the right using the specified node.
130         /// </para>
131         /// <para></para>
132         /// </summary>
133         /// <param name="node">
134         /// <para>The node.</para>
135         /// <para></para>
136         /// </param>
137         /// <param name="right">
138         /// <para>The right.</para>
139         /// <para></para>
140         /// </param>
141         [MethodImpl(MethodImplOptions.AggressiveInlining)]
142         protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
            ↪ LinksIndexParts[node].RightAsSource = right;
143
144         /// <summary>
145         /// <para>
146         /// Gets the size using the specified node.
147         /// </para>
148         /// <para></para>
149         /// </summary>
150         /// <param name="node">
151         /// <para>The node.</para>
152         /// <para></para>
153         /// </param>
154         /// <returns>
155         /// <para>The link</para>
156         /// <para></para>
157         /// </returns>
158         [MethodImpl(MethodImplOptions.AggressiveInlining)]
159         protected override TLinkAddress GetSize(TLinkAddress node) =>
            ↪ LinksIndexParts[node].SizeAsSource;
160
161         /// <summary>
162         /// <para>
163         /// Sets the size using the specified node.
164         /// </para>
165         /// <para></para>
166         /// </summary>
167         /// <param name="node">
168         /// <para>The node.</para>
169         /// <para></para>
170         /// </param>
171         /// <param name="size">
172         /// <para>The size.</para>
173         /// <para></para>
174         /// </param>
175         [MethodImpl(MethodImplOptions.AggressiveInlining)]
176         protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
            ↪ LinksIndexParts[node].SizeAsSource = size;
177
```

```
178         /// <summary>
179         /// <para>
180         /// Gets the tree root using the specified node.
181         /// </para>
182         /// <para></para>
183         /// </summary>
184         /// <param name="node">
185         /// <para>The node.</para>
186         /// <para></para>
187         /// </param>
188         /// <returns>
189         /// <para>The link</para>
190         /// <para></para>
191         /// </returns>
192         [MethodImpl(MethodImplOptions.AggressiveInlining)]
193         protected override TLinkAddress GetTreeRoot(TLinkAddress node) =>
            ↪ LinksIndexParts[node].RootAsSource;
194
195         /// <summary>
196         /// <para>
197         /// Gets the base part value using the specified node.
198         /// </para>
199         /// <para></para>
200         /// </summary>
201         /// <param name="node">
202         /// <para>The node.</para>
203         /// <para></para>
204         /// </param>
205         /// <returns>
206         /// <para>The link</para>
207         /// <para></para>
208         /// </returns>
209         [MethodImpl(MethodImplOptions.AggressiveInlining)]
210         protected override TLinkAddress GetBasePartValue(TLinkAddress node) =>
            ↪ LinksDataParts[node].Source;
211
212         /// <summary>
213         /// <para>
214         /// Gets the key part value using the specified node.
215         /// </para>
216         /// <para></para>
217         /// </summary>
218         /// <param name="node">
219         /// <para>The node.</para>
220         /// <para></para>
221         /// </param>
222         /// <returns>
223         /// <para>The link</para>
224         /// <para></para>
225         /// </returns>
226         [MethodImpl(MethodImplOptions.AggressiveInlining)]
227         protected override TLinkAddress GetKeyPartValue(TLinkAddress node) =>
            ↪ LinksDataParts[node].Target;
228
229         /// <summary>
230         /// <para>
231         /// Clears the node using the specified node.
232         /// </para>
233         /// <para></para>
234         /// </summary>
235         /// <param name="node">
236         /// <para>The node.</para>
237         /// <para></para>
238         /// </param>
239         [MethodImpl(MethodImplOptions.AggressiveInlining)]
240         protected override void ClearNode(TLinkAddress node)
241         {
242             ref var link = ref LinksIndexParts[node];
243             link.LeftAsSource = Zero;
244             link.RightAsSource = Zero;
245             link.SizeAsSource = Zero;
246         }
247
248         /// <summary>
249         /// <para>
250         /// Searches the source.
251         /// </para>
252         /// <para></para>
```

```
253         /// </summary>
254         /// <param name="source">
255         /// <para>The source.</para>
256         /// <para></para>
257         /// </param>
258         /// <param name="target">
259         /// <para>The target.</para>
260         /// <para></para>
261         /// </param>
262         /// <returns>
263         /// <para>The link</para>
264         /// <para></para>
265         /// </returns>
266         public override TLinkAddress Search(TLinkAddress source, TLinkAddress target) =>
            ↪ SearchCore(GetTreeRoot(source), target);
267     }
268 }
```

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

```
1  using System.Runtime.CompilerServices;
2  using TLinkAddress = System.UInt64;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.Split.Specific
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the int 64 internal links sources size balanced tree methods.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     /// <seealso cref="UInt64InternalLinksSizeBalancedTreeMethodsBase"/>
15     public unsafe class UInt64InternalLinksSourcesSizeBalancedTreeMethods :
        ↪ UInt64InternalLinksSizeBalancedTreeMethodsBase
16     {
17         /// <summary>
18         /// <para>
19         /// Initializes a new <see cref="UInt64InternalLinksSourcesSizeBalancedTreeMethods"/>
            ↪ instance.
20         /// </para>
21         /// <para></para>
22         /// </summary>
23         /// <param name="constants">
24         /// <para>A constants.</para>
25         /// <para></para>
26         /// </param>
27         /// <param name="linksDataParts">
28         /// <para>A links data parts.</para>
29         /// <para></para>
30         /// </param>
31         /// <param name="linksIndexParts">
32         /// <para>A links index parts.</para>
33         /// <para></para>
34         /// </param>
35         /// <param name="header">
36         /// <para>A header.</para>
37         /// <para></para>
38         /// </param>
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         public UInt64InternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLinkAddress>
            ↪ constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
            ↪ RawLinkIndexPart<TLinkAddress>* linksIndexParts, LinksHeader<TLinkAddress>* header)
            ↪ : base(constants, linksDataParts, linksIndexParts, header) { }
41
42         /// <summary>
43         /// <para>
44         /// Gets the left reference using the specified node.
45         /// </para>
46         /// <para></para>
47         /// </summary>
48         /// <param name="node">
49         /// <para>The node.</para>
50         /// <para></para>
51         /// </param>
52         /// <returns>
53         /// <para>The ref link</para>
54         /// <para></para>
```

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

        /// <summary>
        /// <para>
        /// Gets the right reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
        ↪ LinksIndexParts[node].RightAsSource;

        /// <summary>
        /// <para>
        /// Gets the left using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetLeft(TLinkAddress node) =>
        ↪ LinksIndexParts[node].LeftAsSource;

        /// <summary>
        /// <para>
        /// Gets the right using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetRight(TLinkAddress node) =>
        ↪ LinksIndexParts[node].RightAsSource;

        /// <summary>
        /// <para>
        /// Sets the left using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="left">
        /// <para>The left.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
        ↪ LinksIndexParts[node].LeftAsSource = left;

```

```
127             /// <summary>
128             /// <para>
129             /// Sets the right using the specified node.
130             /// </para>
131             /// <para></para>
132             /// </summary>
133             /// <param name="node">
134             /// <para>The node.</para>
135             /// <para></para>
136             /// </param>
137             /// <param name="right">
138             /// <para>The right.</para>
139             /// <para></para>
140             /// </param>
141             [MethodImpl(MethodImplOptions.AggressiveInlining)]
142             protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
                ↪ LinksIndexParts[node].RightAsSource = right;
143
144             /// <summary>
145             /// <para>
146             /// Gets the size using the specified node.
147             /// </para>
148             /// <para></para>
149             /// </summary>
150             /// <param name="node">
151             /// <para>The node.</para>
152             /// <para></para>
153             /// </param>
154             /// <returns>
155             /// <para>The link</para>
156             /// <para></para>
157             /// </returns>
158             [MethodImpl(MethodImplOptions.AggressiveInlining)]
159             protected override TLinkAddress GetSize(TLinkAddress node) =>
                ↪ LinksIndexParts[node].SizeAsSource;
160
161             /// <summary>
162             /// <para>
163             /// Sets the size using the specified node.
164             /// </para>
165             /// <para></para>
166             /// </summary>
167             /// <param name="node">
168             /// <para>The node.</para>
169             /// <para></para>
170             /// </param>
171             /// <param name="size">
172             /// <para>The size.</para>
173             /// <para></para>
174             /// </param>
175             [MethodImpl(MethodImplOptions.AggressiveInlining)]
176             protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
                ↪ LinksIndexParts[node].SizeAsSource = size;
177
178             /// <summary>
179             /// <para>
180             /// Gets the tree root using the specified node.
181             /// </para>
182             /// <para></para>
183             /// </summary>
184             /// <param name="node">
185             /// <para>The node.</para>
186             /// <para></para>
187             /// </param>
188             /// <returns>
189             /// <para>The link</para>
190             /// <para></para>
191             /// </returns>
192             [MethodImpl(MethodImplOptions.AggressiveInlining)]
193             protected override TLinkAddress GetTreeRoot(TLinkAddress node) =>
                ↪ LinksIndexParts[node].RootAsSource;
194
195             /// <summary>
196             /// <para>
197             /// Gets the base part value using the specified node.
198             /// </para>
199             /// <para></para>
200             /// </summary>
```

```
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetBasePartValue(TLinkAddress node) =>
        ↪ LinksDataParts[node].Source;

        /// <summary>
        /// <para>
        /// Gets the key part value using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetKeyPartValue(TLinkAddress node) =>
        ↪ LinksDataParts[node].Target;

        /// <summary>
        /// <para>
        /// Clears the node using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLinkAddress node)
        {
            ref var link = ref LinksIndexParts[node];
            link.LeftAsSource = Zero;
            link.RightAsSource = Zero;
            link.SizeAsSource = Zero;
        }

        /// <summary>
        /// <para>
        /// Searches the source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public override TLinkAddress Search(TLinkAddress source, TLinkAddress target) =>
        ↪ SearchCore(GetTreeRoot(source), target);
    }
}
```

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

```
using System.Runtime.CompilerServices;
using TLinkAddress = System.UInt64;

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

```

```
6  namespace Platform.Data.Doublets.Memory.Split.Specific
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the int 64 internal links targets recursionless size balanced tree methods.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     /// <seealso cref="UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase"/>
15     public unsafe class UInt64InternalLinksTargetsRecursionlessSizeBalancedTreeMethods :
       ↪  UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase
16     {
17         /// <summary>
18         /// <para>
19         /// Initializes a new <see
           ↪  cref="UInt64InternalLinksTargetsRecursionlessSizeBalancedTreeMethods"/> instance.
20         /// </para>
21         /// <para></para>
22         /// </summary>
23         /// <param name="constants">
24         /// <para>A constants.</para>
25         /// <para></para>
26         /// </param>
27         /// <param name="linksDataParts">
28         /// <para>A links data parts.</para>
29         /// <para></para>
30         /// </param>
31         /// <param name="linksIndexParts">
32         /// <para>A links index parts.</para>
33         /// <para></para>
34         /// </param>
35         /// <param name="header">
36         /// <para>A header.</para>
37         /// <para></para>
38         /// </param>
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         public UInt64InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<TLi
           ↪  nkAddress> constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
           ↪  RawLinkIndexPart<TLinkAddress>* linksIndexParts, LinksHeader<TLinkAddress>* header)
           ↪  : base(constants, linksDataParts, linksIndexParts, header) { }
41
42         /// <summary>
43         /// <para>
44         /// Gets the left reference using the specified node.
45         /// </para>
46         /// <para></para>
47         /// </summary>
48         /// <param name="node">
49         /// <para>The node.</para>
50         /// <para></para>
51         /// </param>
52         /// <returns>
53         /// <para>The ref ulong</para>
54         /// <para></para>
55         /// </returns>
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         protected override ref ulong GetLeftReference(ulong node) => ref
           ↪  LinksIndexParts[node].LeftAsTarget;
58
59         /// <summary>
60         /// <para>
61         /// Gets the right reference using the specified node.
62         /// </para>
63         /// <para></para>
64         /// </summary>
65         /// <param name="node">
66         /// <para>The node.</para>
67         /// <para></para>
68         /// </param>
69         /// <returns>
70         /// <para>The ref ulong</para>
71         /// <para></para>
72         /// </returns>
73         [MethodImpl(MethodImplOptions.AggressiveInlining)]
74         protected override ref ulong GetRightReference(ulong node) => ref
           ↪  LinksIndexParts[node].RightAsTarget;
75
76         /// <summary>
```

```
        /// <para>
        /// Gets the left using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetLeft(TLinkAddress node) =>
        ↪ LinksIndexParts[node].LeftAsTarget;

        /// <summary>
        /// <para>
        /// Gets the right using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetRight(TLinkAddress node) =>
        ↪ LinksIndexParts[node].RightAsTarget;

        /// <summary>
        /// <para>
        /// Sets the left using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="left">
        /// <para>The left.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
        ↪ LinksIndexParts[node].LeftAsTarget = left;

        /// <summary>
        /// <para>
        /// Sets the right using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="right">
        /// <para>The right.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
        ↪ LinksIndexParts[node].RightAsTarget = right;

        /// <summary>
        /// <para>
        /// Gets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
```

```
151         /// <para>The node.</para>
152         /// <para></para>
153         /// </param>
154         /// <returns>
155         /// <para>The link</para>
156         /// <para></para>
157         /// </returns>
158         [MethodImpl(MethodImplOptions.AggressiveInlining)]
159         protected override TLinkAddress GetSize(TLinkAddress node) =>
            ↪ LinksIndexParts[node].SizeAsTarget;
160
161         /// <summary>
162         /// <para>
163         /// Sets the size using the specified node.
164         /// </para>
165         /// <para></para>
166         /// </summary>
167         /// <param name="node">
168         /// <para>The node.</para>
169         /// <para></para>
170         /// </param>
171         /// <param name="size">
172         /// <para>The size.</para>
173         /// <para></para>
174         /// </param>
175         [MethodImpl(MethodImplOptions.AggressiveInlining)]
176         protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
            ↪ LinksIndexParts[node].SizeAsTarget = size;
177
178         /// <summary>
179         /// <para>
180         /// Gets the tree root using the specified node.
181         /// </para>
182         /// <para></para>
183         /// </summary>
184         /// <param name="node">
185         /// <para>The node.</para>
186         /// <para></para>
187         /// </param>
188         /// <returns>
189         /// <para>The link</para>
190         /// <para></para>
191         /// </returns>
192         [MethodImpl(MethodImplOptions.AggressiveInlining)]
193         protected override TLinkAddress GetTreeRoot(TLinkAddress node) =>
            ↪ LinksIndexParts[node].RootAsTarget;
194
195         /// <summary>
196         /// <para>
197         /// Gets the base part value using the specified node.
198         /// </para>
199         /// <para></para>
200         /// </summary>
201         /// <param name="node">
202         /// <para>The node.</para>
203         /// <para></para>
204         /// </param>
205         /// <returns>
206         /// <para>The link</para>
207         /// <para></para>
208         /// </returns>
209         [MethodImpl(MethodImplOptions.AggressiveInlining)]
210         protected override TLinkAddress GetBasePartValue(TLinkAddress node) =>
            ↪ LinksDataParts[node].Target;
211
212         /// <summary>
213         /// <para>
214         /// Gets the key part value using the specified node.
215         /// </para>
216         /// <para></para>
217         /// </summary>
218         /// <param name="node">
219         /// <para>The node.</para>
220         /// <para></para>
221         /// </param>
222         /// <returns>
223         /// <para>The link</para>
224         /// <para></para>
```

```
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetKeyPartValue(TLinkAddress node) =>
        ↪ LinksDataParts[node].Source;

        /// <summary>
        /// <para>
        /// Clears the node using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLinkAddress node)
        {
            ref var link = ref LinksIndexParts[node];
            link.LeftAsTarget = Zero;
            link.RightAsTarget = Zero;
            link.SizeAsTarget = Zero;
        }

        /// <summary>
        /// <para>
        /// Searches the source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public override TLinkAddress Search(TLinkAddress source, TLinkAddress target) =>
        ↪ SearchCore(GetTreeRoot(target), source);
    }
}
```

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

```
using System.Runtime.CompilerServices;
using TLinkAddress = System.UInt64;

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    /// <summary>
    /// <para>
    /// Represents the int 64 internal links targets size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="UInt64InternalLinksSizeBalancedTreeMethodsBase"/>
    public unsafe class UInt64InternalLinksTargetsSizeBalancedTreeMethods :
    ↪ UInt64InternalLinksSizeBalancedTreeMethodsBase
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt64InternalLinksTargetsSizeBalancedTreeMethods"/>
        ↪ instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="linksDataParts">
        /// <para>A links data parts.</para>
```

```
        /// <para></para>
        /// </param>
        /// <param name="linksIndexParts">
        /// <para>A links index parts.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64InternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLinkAddress>
        ↪  constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
        ↪  RawLinkIndexPart<TLinkAddress>* linksIndexParts, LinksHeader<TLinkAddress>* header)
        ↪  : base(constants, linksDataParts, linksIndexParts, header) { }

        /// <summary>
        /// <para>
        /// Gets the left reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetLeftReference(ulong node) => ref
        ↪  LinksIndexParts[node].LeftAsTarget;

        /// <summary>
        /// <para>
        /// Gets the right reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetRightReference(ulong node) => ref
        ↪  LinksIndexParts[node].RightAsTarget;

        /// <summary>
        /// <para>
        /// Gets the left using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetLeft(TLinkAddress node) =>
        ↪  LinksIndexParts[node].LeftAsTarget;

        /// <summary>
        /// <para>
        /// Gets the right using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
```

```
101             /// <para></para>
102             /// </param>
103             /// <returns>
104             /// <para>The link</para>
105             /// <para></para>
106             /// </returns>
107             [MethodImpl(MethodImplOptions.AggressiveInlining)]
108             protected override TLinkAddress GetRight(TLinkAddress node) =>
                ↪ LinksIndexParts[node].RightAsTarget;
109
110             /// <summary>
111             /// <para>
112             /// Sets the left using the specified node.
113             /// </para>
114             /// <para></para>
115             /// </summary>
116             /// <param name="node">
117             /// <para>The node.</para>
118             /// <para></para>
119             /// </param>
120             /// <param name="left">
121             /// <para>The left.</para>
122             /// <para></para>
123             /// </param>
124             [MethodImpl(MethodImplOptions.AggressiveInlining)]
125             protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
                ↪ LinksIndexParts[node].LeftAsTarget = left;
126
127             /// <summary>
128             /// <para>
129             /// Sets the right using the specified node.
130             /// </para>
131             /// <para></para>
132             /// </summary>
133             /// <param name="node">
134             /// <para>The node.</para>
135             /// <para></para>
136             /// </param>
137             /// <param name="right">
138             /// <para>The right.</para>
139             /// <para></para>
140             /// </param>
141             [MethodImpl(MethodImplOptions.AggressiveInlining)]
142             protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
                ↪ LinksIndexParts[node].RightAsTarget = right;
143
144             /// <summary>
145             /// <para>
146             /// Gets the size using the specified node.
147             /// </para>
148             /// <para></para>
149             /// </summary>
150             /// <param name="node">
151             /// <para>The node.</para>
152             /// <para></para>
153             /// </param>
154             /// <returns>
155             /// <para>The link</para>
156             /// <para></para>
157             /// </returns>
158             [MethodImpl(MethodImplOptions.AggressiveInlining)]
159             protected override TLinkAddress GetSize(TLinkAddress node) =>
                ↪ LinksIndexParts[node].SizeAsTarget;
160
161             /// <summary>
162             /// <para>
163             /// Sets the size using the specified node.
164             /// </para>
165             /// <para></para>
166             /// </summary>
167             /// <param name="node">
168             /// <para>The node.</para>
169             /// <para></para>
170             /// </param>
171             /// <param name="size">
172             /// <para>The size.</para>
173             /// <para></para>
174             /// </param>
```

```
175             [MethodImpl(MethodImplOptions.AggressiveInlining)]
176             protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
                ↪ LinksIndexParts[node].SizeAsTarget = size;
177
178             /// <summary>
179             /// <para>
180             /// Gets the tree root using the specified node.
181             /// </para>
182             /// <para></para>
183             /// </summary>
184             /// <param name="node">
185             /// <para>The node.</para>
186             /// <para></para>
187             /// </param>
188             /// <returns>
189             /// <para>The link</para>
190             /// <para></para>
191             /// </returns>
192             [MethodImpl(MethodImplOptions.AggressiveInlining)]
193             protected override TLinkAddress GetTreeRoot(TLinkAddress node) =>
                ↪ LinksIndexParts[node].RootAsTarget;
194
195             /// <summary>
196             /// <para>
197             /// Gets the base part value using the specified node.
198             /// </para>
199             /// <para></para>
200             /// </summary>
201             /// <param name="node">
202             /// <para>The node.</para>
203             /// <para></para>
204             /// </param>
205             /// <returns>
206             /// <para>The link</para>
207             /// <para></para>
208             /// </returns>
209             [MethodImpl(MethodImplOptions.AggressiveInlining)]
210             protected override TLinkAddress GetBasePartValue(TLinkAddress node) =>
                ↪ LinksDataParts[node].Target;
211
212             /// <summary>
213             /// <para>
214             /// Gets the key part value using the specified node.
215             /// </para>
216             /// <para></para>
217             /// </summary>
218             /// <param name="node">
219             /// <para>The node.</para>
220             /// <para></para>
221             /// </param>
222             /// <returns>
223             /// <para>The link</para>
224             /// <para></para>
225             /// </returns>
226             [MethodImpl(MethodImplOptions.AggressiveInlining)]
227             protected override TLinkAddress GetKeyPartValue(TLinkAddress node) =>
                ↪ LinksDataParts[node].Source;
228
229             /// <summary>
230             /// <para>
231             /// Clears the node using the specified node.
232             /// </para>
233             /// <para></para>
234             /// </summary>
235             /// <param name="node">
236             /// <para>The node.</para>
237             /// <para></para>
238             /// </param>
239             [MethodImpl(MethodImplOptions.AggressiveInlining)]
240             protected override void ClearNode(TLinkAddress node)
241             {
242                 ref var link = ref LinksIndexParts[node];
243                 link.LeftAsTarget = Zero;
244                 link.RightAsTarget = Zero;
245                 link.SizeAsTarget = Zero;
246             }
247
248             /// <summary>
```

```
249         /// <para>
250         /// Searches the source.
251         /// </para>
252         /// <para></para>
253         /// </summary>
254         /// <param name="source">
255         /// <para>The source.</para>
256         /// <para></para>
257         /// </param>
258         /// <param name="target">
259         /// <para>The target.</para>
260         /// <para></para>
261         /// </param>
262         /// <returns>
263         /// <para>The link</para>
264         /// <para></para>
265         /// </returns>
266         public override TLinkAddress Search(TLinkAddress source, TLinkAddress target) =>
            ↪ SearchCore(GetTreeRoot(target), source);
267     }
268 }
```

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

```
1  using System;
2  using System.Runtime.CompilerServices;
3  using Platform.Singletons;
4  using Platform.Memory;
5  using Platform.Data.Doublets.Memory.Split.Generic;
6  using TLinkAddress = System.UInt64;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Memory.Split.Specific
11 {
12     /// <summary>
13     /// <para>
14     /// Represents the int 64 split memory links.
15     /// </para>
16     /// <para></para>
17     /// </summary>
18     /// <seealso cref="SplitMemoryLinksBase{TLinkAddress}"/>
19     public unsafe class UInt64SplitMemoryLinks : SplitMemoryLinksBase<TLinkAddress>
20     {
21         private readonly Func<ILinksTreeMethods<TLinkAddress>> _createInternalSourceTreeMethods;
22         private readonly Func<ILinksTreeMethods<TLinkAddress>> _createExternalSourceTreeMethods;
23         private readonly Func<ILinksTreeMethods<TLinkAddress>> _createInternalTargetTreeMethods;
24         private readonly Func<ILinksTreeMethods<TLinkAddress>> _createExternalTargetTreeMethods;
25         private LinksHeader<ulong>* _header;
26         private RawLinkDataPart<ulong>* _linksDataParts;
27         private RawLinkIndexPart<ulong>* _linksIndexParts;
28
29         /// <summary>
30         /// <para>
31         /// Initializes a new <see cref="UInt64SplitMemoryLinks"/> instance.
32         /// </para>
33         /// <para></para>
34         /// </summary>
35         /// <param name="dataMemory">
36         /// <para>A data memory.</para>
37         /// <para></para>
38         /// </param>
39         /// <param name="indexMemory">
40         /// <para>A index memory.</para>
41         /// <para></para>
42         /// </param>
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         public UInt64SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪ indexMemory) : this(dataMemory, indexMemory, DefaultLinksSizeStep) { }
45
46         /// <summary>
47         /// <para>
48         /// Initializes a new <see cref="UInt64SplitMemoryLinks"/> instance.
49         /// </para>
50         /// <para></para>
51         /// </summary>
52         /// <param name="dataMemory">
53         /// <para>A data memory.</para>
54         /// <para></para>
55         /// </param>
```

```
56          /// <param name="indexMemory">
57          /// <para>A index memory.</para>
58          /// <para></para>
59          /// </param>
60          /// <param name="memoryReservationStep">
61          /// <para>A memory reservation step.</para>
62          /// <para></para>
63          /// </param>
64          [MethodImpl(MethodImplOptions.AggressiveInlining)]
65          public UInt64SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪  indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
            ↪  memoryReservationStep, Default<LinksConstants<TLinkAddress>>.Instance,
            ↪  IndexTreeType.Default, useLinkedList: true) { }
66
67          /// <summary>
68          /// <para>
69          /// Initializes a new <see cref="UInt64SplitMemoryLinks"/> instance.
70          /// </para>
71          /// <para></para>
72          /// </summary>
73          /// <param name="dataMemory">
74          /// <para>A data memory.</para>
75          /// <para></para>
76          /// </param>
77          /// <param name="indexMemory">
78          /// <para>A index memory.</para>
79          /// <para></para>
80          /// </param>
81          /// <param name="memoryReservationStep">
82          /// <para>A memory reservation step.</para>
83          /// <para></para>
84          /// </param>
85          /// <param name="constants">
86          /// <para>A constants.</para>
87          /// <para></para>
88          /// </param>
89          [MethodImpl(MethodImplOptions.AggressiveInlining)]
90          public UInt64SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪  indexMemory, long memoryReservationStep, LinksConstants<TLinkAddress> constants) :
            ↪  this(dataMemory, indexMemory, memoryReservationStep, constants,
            ↪  IndexTreeType.Default, useLinkedList: true) { }
91
92          /// <summary>
93          /// <para>
94          /// Initializes a new <see cref="UInt64SplitMemoryLinks"/> instance.
95          /// </para>
96          /// <para></para>
97          /// </summary>
98          /// <param name="dataMemory">
99          /// <para>A data memory.</para>
100         /// <para></para>
101         /// </param>
102         /// <param name="indexMemory">
103         /// <para>A index memory.</para>
104         /// <para></para>
105         /// </param>
106         /// <param name="memoryReservationStep">
107         /// <para>A memory reservation step.</para>
108         /// <para></para>
109         /// </param>
110         /// <param name="constants">
111         /// <para>A constants.</para>
112         /// <para></para>
113         /// </param>
114         /// <param name="indexTreeType">
115         /// <para>A index tree type.</para>
116         /// <para></para>
117         /// </param>
118         /// <param name="useLinkedList">
119         /// <para>A use linked list.</para>
120         /// <para></para>
121         /// </param>
122         [MethodImpl(MethodImplOptions.AggressiveInlining)]
123         public UInt64SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪  indexMemory, long memoryReservationStep, LinksConstants<TLinkAddress> constants,
            ↪  IndexTreeType indexTreeType, bool useLinkedList) : base(dataMemory, indexMemory,
            ↪  memoryReservationStep, constants, useLinkedList)
```

```
124         {
125             if (indexTreeType == IndexTreeType.SizeBalancedTree)
126             {
127                 _createInternalSourceTreeMethods = () => new
                    ↪ UInt64InternalLinksSourcesSizeBalancedTreeMethods(Constants,
                    ↪ _linksDataParts, _linksIndexParts, _header);
128                 _createExternalSourceTreeMethods = () => new
                    ↪ UInt64ExternalLinksSourcesSizeBalancedTreeMethods(Constants,
                    ↪ _linksDataParts, _linksIndexParts, _header);
129                 _createInternalTargetTreeMethods = () => new
                    ↪ UInt64InternalLinksTargetsSizeBalancedTreeMethods(Constants,
                    ↪ _linksDataParts, _linksIndexParts, _header);
130                 _createExternalTargetTreeMethods = () => new
                    ↪ UInt64ExternalLinksTargetsSizeBalancedTreeMethods(Constants,
                    ↪ _linksDataParts, _linksIndexParts, _header);
131             }
132             else
133             {
134                 _createInternalSourceTreeMethods = () => new
                    ↪ UInt64InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(Constants,
                    ↪ _linksDataParts, _linksIndexParts, _header);
135                 _createExternalSourceTreeMethods = () => new
                    ↪ UInt64ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(Constants,
                    ↪ _linksDataParts, _linksIndexParts, _header);
136                 _createInternalTargetTreeMethods = () => new
                    ↪ UInt64InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(Constants,
                    ↪ _linksDataParts, _linksIndexParts, _header);
137                 _createExternalTargetTreeMethods = () => new
                    ↪ UInt64ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(Constants,
                    ↪ _linksDataParts, _linksIndexParts, _header);
138             }
139             Init(dataMemory, indexMemory);
140         }
141 
142         /// <summary>
143         /// <para>
144         /// Sets the pointers using the specified data memory.
145         /// </para>
146         /// <para></para>
147         /// </summary>
148         /// <param name="dataMemory">
149         /// <para>The data memory.</para>
150         /// <para></para>
151         /// </param>
152         /// <param name="indexMemory">
153         /// <para>The index memory.</para>
154         /// <para></para>
155         /// </param>
156         [MethodImpl(MethodImplOptions.AggressiveInlining)]
157         protected override void SetPointers(IResizableDirectMemory dataMemory,
            ↪ IResizableDirectMemory indexMemory)
158         {
159             _linksDataParts = (RawLinkDataPart<TLinkAddress>*)dataMemory.Pointer;
160             _linksIndexParts = (RawLinkIndexPart<TLinkAddress>*)indexMemory.Pointer;
161             _header = (LinksHeader<TLinkAddress>*)indexMemory.Pointer;
162             if (_useLinkedList)
163             {
164                 InternalSourcesListMethods = new
                    ↪ UInt64InternalLinksSourcesLinkedListMethods(Constants, _linksDataParts,
                    ↪ _linksIndexParts);
165             }
166             else
167             {
168                 InternalSourcesTreeMethods = _createInternalSourceTreeMethods();
169             }
170             ExternalSourcesTreeMethods = _createExternalSourceTreeMethods();
171             InternalTargetsTreeMethods = _createInternalTargetTreeMethods();
172             ExternalTargetsTreeMethods = _createExternalTargetTreeMethods();
173             UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_linksDataParts, _header);
174         }
175 
176         /// <summary>
177         /// <para>
178         /// Resets the pointers.
179         /// </para>
180         /// <para></para>
181         /// </summary>
```

```
182         [MethodImpl(MethodImplOptions.AggressiveInlining)]
183         protected override void ResetPointers()
184         {
185             base.ResetPointers();
186             _linksDataParts = null;
187             _linksIndexParts = null;
188             _header = null;
189         }
190 
191         /// <summary>
192         /// <para>
193         /// Gets the header reference.
194         /// </para>
195         /// <para></para>
196         /// </summary>
197         /// <returns>
198         /// <para>A ref links header of t link</para>
199         /// <para></para>
200         /// </returns>
201         [MethodImpl(MethodImplOptions.AggressiveInlining)]
202         protected override ref LinksHeader<TLinkAddress> GetHeaderReference() => ref *_header;
203 
204         /// <summary>
205         /// <para>
206         /// Gets the link data part reference using the specified link index.
207         /// </para>
208         /// <para></para>
209         /// </summary>
210         /// <param name="linkIndex">
211         /// <para>The link index.</para>
212         /// <para></para>
213         /// </param>
214         /// <returns>
215         /// <para>A ref raw link data part of t link</para>
216         /// <para></para>
217         /// </returns>
218         [MethodImpl(MethodImplOptions.AggressiveInlining)]
219         protected override ref RawLinkDataPart<TLinkAddress>
            ↪ GetLinkDataPartReference(TLinkAddress linkIndex) => ref _linksDataParts[linkIndex];
220 
221         /// <summary>
222         /// <para>
223         /// Gets the link index part reference using the specified link index.
224         /// </para>
225         /// <para></para>
226         /// </summary>
227         /// <param name="linkIndex">
228         /// <para>The link index.</para>
229         /// <para></para>
230         /// </param>
231         /// <returns>
232         /// <para>A ref raw link index part of t link</para>
233         /// <para></para>
234         /// </returns>
235         [MethodImpl(MethodImplOptions.AggressiveInlining)]
236         protected override ref RawLinkIndexPart<TLinkAddress>
            ↪ GetLinkIndexPartReference(TLinkAddress linkIndex) => ref _linksIndexParts[linkIndex];
237 
238         /// <summary>
239         /// <para>
240         /// Determines whether this instance are equal.
241         /// </para>
242         /// <para></para>
243         /// </summary>
244         /// <param name="first">
245         /// <para>The first.</para>
246         /// <para></para>
247         /// </param>
248         /// <param name="second">
249         /// <para>The second.</para>
250         /// <para></para>
251         /// </param>
252         /// <returns>
253         /// <para>The bool</para>
254         /// <para></para>
255         /// </returns>
256         [MethodImpl(MethodImplOptions.AggressiveInlining)]
257         protected override bool AreEqual(ulong first, ulong second) => first == second;
```

```
        /// <summary>
        /// <para>
        /// Determines whether this instance less than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessThan(ulong first, ulong second) => first < second;

        /// <summary>
        /// <para>
        /// Determines whether this instance less or equal than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThan(ulong first, ulong second) => first > second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater or equal than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
```

```
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;

        /// <summary>
        /// <para>
        /// Gets the zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetZero() => 0UL;

        /// <summary>
        /// <para>
        /// Gets the one.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetOne() => 1UL;

        /// <summary>
        /// <para>
        /// Converts the to int 64 using the specified value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The long</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override long ConvertToInt64(ulong value) => (long)value;

        /// <summary>
        /// <para>
        /// Converts the to address using the specified value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong ConvertToAddress(long value) => (ulong)value;

        /// <summary>
        /// <para>
        /// Adds the first.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
```

```
414             /// <para>The second.</para>
415             /// <para></para>
416             /// </param>
417             /// <returns>
418             /// <para>The ulong</para>
419             /// <para></para>
420             /// </returns>
421             [MethodImpl(MethodImplOptions.AggressiveInlining)]
422             protected override ulong Add(ulong first, ulong second) => first + second;
423
424             /// <summary>
425             /// <para>
426             /// Subtracts the first.
427             /// </para>
428             /// <para></para>
429             /// </summary>
430             /// <param name="first">
431             /// <para>The first.</para>
432             /// <para></para>
433             /// </param>
434             /// <param name="second">
435             /// <para>The second.</para>
436             /// <para></para>
437             /// </param>
438             /// <returns>
439             /// <para>The ulong</para>
440             /// <para></para>
441             /// </returns>
442             [MethodImpl(MethodImplOptions.AggressiveInlining)]
443             protected override ulong Subtract(ulong first, ulong second) => first - second;
444
445             /// <summary>
446             /// <para>
447             /// Increments the link.
448             /// </para>
449             /// <para></para>
450             /// </summary>
451             /// <param name="link">
452             /// <para>The link.</para>
453             /// <para></para>
454             /// </param>
455             /// <returns>
456             /// <para>The ulong</para>
457             /// <para></para>
458             /// </returns>
459             [MethodImpl(MethodImplOptions.AggressiveInlining)]
460             protected override ulong Increment(ulong link) => ++link;
461
462             /// <summary>
463             /// <para>
464             /// Decrements the link.
465             /// </para>
466             /// <para></para>
467             /// </summary>
468             /// <param name="link">
469             /// <para>The link.</para>
470             /// <para></para>
471             /// </param>
472             /// <returns>
473             /// <para>The ulong</para>
474             /// <para></para>
475             /// </returns>
476             [MethodImpl(MethodImplOptions.AggressiveInlining)]
477             protected override ulong Decrement(ulong link) => --link;
478         }
479     }
```

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

```
1   using System.Runtime.CompilerServices;
2   using Platform.Data.Doublets.Memory.Split.Generic;
3   using TLinkAddress = System.UInt64;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.Memory.Split.Specific
8   {
9       /// <summary>
10      /// <para>
```

```
    /// Represents the int 64 unused links list methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="UnusedLinksListMethods{TLinkAddress}"/>
    public unsafe class UInt64UnusedLinksListMethods : UnusedLinksListMethods<TLinkAddress>
    {
        private readonly RawLinkDataPart<ulong>* _links;
        private readonly LinksHeader<ulong>* _header;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt64UnusedLinksListMethods"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64UnusedLinksListMethods(RawLinkDataPart<ulong>* links, LinksHeader<ulong>*
        ↪  header)
            : base((byte*)links, (byte*)header)
        {
            _links = links;
            _header = header;
        }

        /// <summary>
        /// <para>
        /// Gets the link data part reference using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link data part of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkDataPart<TLinkAddress>
        ↪  GetLinkDataPartReference(TLinkAddress link) => ref _links[link];

        /// <summary>
        /// <para>
        /// Gets the header reference.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A ref links header of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref LinksHeader<TLinkAddress> GetHeaderReference() => ref *_header;
    }
}
```

1.80 ./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.Delegates;
using Platform.Numbers;
using static System.Runtime.CompilerServices.Unsafe;

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

```

```
13  namespace Platform.Data.Doublets.Memory.United.Generic
14  {
15      /// <summary>
16      /// <para>
17      /// Represents the links avl balanced tree methods base.
18      /// </para>
19      /// <para></para>
20      /// </summary>
21      /// <seealso cref="SizedAndThreadedAVLBalancedTreeMethods{TLinkAddress}"/>
22      /// <seealso cref="ILinksTreeMethods{TLinkAddress}"/>
23      public unsafe abstract class LinksAvlBalancedTreeMethodsBase<TLinkAddress> :
        ↪  SizedAndThreadedAVLBalancedTreeMethods<TLinkAddress>, ILinksTreeMethods<TLinkAddress>
24      {
25          private static readonly UncheckedConverter<TLinkAddress, long> _addressToInt64Converter
            ↪  = UncheckedConverter<TLinkAddress, long>.Default;
26          private static readonly UncheckedConverter<TLinkAddress, int> _addressToInt32Converter =
            ↪  UncheckedConverter<TLinkAddress, int>.Default;
27          private static readonly UncheckedConverter<bool, TLinkAddress> _boolToAddressConverter =
            ↪  UncheckedConverter<bool, TLinkAddress>.Default;
28          private static readonly UncheckedConverter<TLinkAddress, bool> _addressToBoolConverter =
            ↪  UncheckedConverter<TLinkAddress, bool>.Default;
29          private static readonly UncheckedConverter<int, TLinkAddress> _int32ToAddressConverter =
            ↪  UncheckedConverter<int, TLinkAddress>.Default;
30
31          /// <summary>
32          /// <para>
33          /// The break.
34          /// </para>
35          /// <para></para>
36          /// </summary>
37          protected readonly TLinkAddress Break;
38          /// <summary>
39          /// <para>
40          /// The continue.
41          /// </para>
42          /// <para></para>
43          /// </summary>
44          protected readonly TLinkAddress Continue;
45          /// <summary>
46          /// <para>
47          /// The links.
48          /// </para>
49          /// <para></para>
50          /// </summary>
51          protected readonly byte* Links;
52          /// <summary>
53          /// <para>
54          /// The header.
55          /// </para>
56          /// <para></para>
57          /// </summary>
58          protected readonly byte* Header;
59
60          /// <summary>
61          /// <para>
62          /// Initializes a new <see cref="LinksAvlBalancedTreeMethodsBase"/> instance.
63          /// </para>
64          /// <para></para>
65          /// </summary>
66          /// <param name="constants">
67          /// <para>A constants.</para>
68          /// <para></para>
69          /// </param>
70          /// <param name="links">
71          /// <para>A links.</para>
72          /// <para></para>
73          /// </param>
74          /// <param name="header">
75          /// <para>A header.</para>
76          /// <para></para>
77          /// </param>
78          [MethodImpl(MethodImplOptions.AggressiveInlining)]
79          protected LinksAvlBalancedTreeMethodsBase(LinksConstants<TLinkAddress> constants, byte*
            ↪  links, byte* header)
80          {
81              Links = links;
82              Header = header;
83              Break = constants.Break;
84              Continue = constants.Continue;
```

```
        }

        /// <summary>
        /// <para>
        /// Gets the tree root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TLinkAddress GetTreeRoot();

        /// <summary>
        /// <para>
        /// Gets the base part value using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TLinkAddress GetBasePartValue(TLinkAddress link);

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the right of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <param name="rootSource">
        /// <para>The root source.</para>
        /// <para></para>
        /// </param>
        /// <param name="rootTarget">
        /// <para>The root target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract bool FirstIsToTheRightOfSecond(TLinkAddress source, TLinkAddress
        ↪  target, TLinkAddress rootSource, TLinkAddress rootTarget);

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <param name="rootSource">
        /// <para>The root source.</para>
```

```
        /// <para></para>
        /// </param>
        /// <param name="rootTarget">
        /// <para>The root target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract bool FirstIsToTheLeftOfSecond(TLinkAddress source, TLinkAddress
        ↪  target, TLinkAddress rootSource, TLinkAddress rootTarget);

        /// <summary>
        /// <para>
        /// Gets the header reference.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A ref links header of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref LinksHeader<TLinkAddress> GetHeaderReference() => ref
        ↪  AsRef<LinksHeader<TLinkAddress>>(Header);

        /// <summary>
        /// <para>
        /// Gets the link reference using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref RawLink<TLinkAddress> GetLinkReference(TLinkAddress link) => ref
        ↪  AsRef<RawLink<TLinkAddress>>(Links + (RawLink<TLinkAddress>.SizeInBytes *
        ↪  _addressToInt64Converter.Convert(link)));

        /// <summary>
        /// <para>
        /// Gets the link values using the specified link index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="linkIndex">
        /// <para>The link index.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A list of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual IList<TLinkAddress>? GetLinkValues(TLinkAddress linkIndex)
        {
            ref var link = ref GetLinkReference(linkIndex);
            return new Link<TLinkAddress>(linkIndex, link.Source, link.Target);
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
```

```
236         /// <param name="second">
237         /// <para>The second.</para>
238         /// <para></para>
239         /// </param>
240         /// <returns>
241         /// <para>The bool</para>
242         /// <para></para>
243         /// </returns>
244         [MethodImpl(MethodImplOptions.AggressiveInlining)]
245         protected override bool FirstIsToTheLeftOfSecond(TLinkAddress first, TLinkAddress second)
246         {
247             ref var firstLink = ref GetLinkReference(first);
248             ref var secondLink = ref GetLinkReference(second);
249             return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
250         }
251 
252         /// <summary>
253         /// <para>
254         /// Determines whether this instance first is to the right of second.
255         /// </para>
256         /// <para></para>
257         /// </summary>
258         /// <param name="first">
259         /// <para>The first.</para>
260         /// <para></para>
261         /// </param>
262         /// <param name="second">
263         /// <para>The second.</para>
264         /// <para></para>
265         /// </param>
266         /// <returns>
267         /// <para>The bool</para>
268         /// <para></para>
269         /// </returns>
270         [MethodImpl(MethodImplOptions.AggressiveInlining)]
271         protected override bool FirstIsToTheRightOfSecond(TLinkAddress first, TLinkAddress
            ↪  second)
272         {
273             ref var firstLink = ref GetLinkReference(first);
274             ref var secondLink = ref GetLinkReference(second);
275             return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
276         }
277 
278         /// <summary>
279         /// <para>
280         /// Gets the size value using the specified value.
281         /// </para>
282         /// <para></para>
283         /// </summary>
284         /// <param name="value">
285         /// <para>The value.</para>
286         /// <para></para>
287         /// </param>
288         /// <returns>
289         /// <para>The link</para>
290         /// <para></para>
291         /// </returns>
292         [MethodImpl(MethodImplOptions.AggressiveInlining)]
293         protected virtual TLinkAddress GetSizeValue(TLinkAddress value) =>
            ↪  Bit<TLinkAddress>.PartialRead(value, 5, -5);
294 
295         /// <summary>
296         /// <para>
297         /// Sets the size value using the specified stored value.
298         /// </para>
299         /// <para></para>
300         /// </summary>
301         /// <param name="storedValue">
302         /// <para>The stored value.</para>
303         /// <para></para>
304         /// </param>
305         /// <param name="size">
306         /// <para>The size.</para>
307         /// <para></para>
308         /// </param>
```

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

        /// <summary>
        /// <para>
        /// Determines whether this instance get left is child value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool GetLeftIsChildValue(TLinkAddress value)
        {
            unchecked
            {
                return _addressToBoolConverter.Convert(Bit<TLinkAddress>.PartialRead(value, 4,
                ↪  1));
                //return !EqualityComparer.Equals(Bit<TLinkAddress>.PartialRead(value, 4, 1),
                ↪  default);
            }
        }

        /// <summary>
        /// <para>
        /// Sets the left is child value using the specified stored value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="storedValue">
        /// <para>The stored value.</para>
        /// <para></para>
        /// </param>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual void SetLeftIsChildValue(ref TLinkAddress storedValue, bool value)
        {
            unchecked
            {
                var previousValue = storedValue;
                var modified = Bit<TLinkAddress>.PartialWrite(previousValue,
                ↪  _boolToAddressConverter.Convert(value), 4, 1);
                storedValue = modified;
            }
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance get right is child value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool GetRightIsChildValue(TLinkAddress value)
        {
            unchecked
            {
                return _addressToBoolConverter.Convert(Bit<TLinkAddress>.PartialRead(value, 3,
                ↪  1));
```

```
            //return !EqualityComparer.Equals(Bit<TLinkAddress>.PartialRead(value, 3, 1),
            ↪  default);
        }
    }

    /// <summary>
    /// <para>
    /// Sets the right is child value using the specified stored value.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <param name="storedValue">
    /// <para>The stored value.</para>
    /// <para></para>
    /// </param>
    /// <param name="value">
    /// <para>The value.</para>
    /// <para></para>
    /// </param>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    protected virtual void SetRightIsChildValue(ref TLinkAddress storedValue, bool value)
    {
        unchecked
        {
            var previousValue = storedValue;
            var modified = Bit<TLinkAddress>.PartialWrite(previousValue,
            ↪  _boolToAddressConverter.Convert(value), 3, 1);
            storedValue = modified;
        }
    }

    /// <summary>
    /// <para>
    /// Determines whether this instance is child.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <param name="parent">
    /// <para>The parent.</para>
    /// <para></para>
    /// </param>
    /// <param name="possibleChild">
    /// <para>The possible child.</para>
    /// <para></para>
    /// </param>
    /// <returns>
    /// <para>The bool</para>
    /// <para></para>
    /// </returns>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    protected bool IsChild(TLinkAddress parent, TLinkAddress possibleChild)
    {
        var parentSize = GetSize(parent);
        var childSize = GetSizeOrZero(possibleChild);
        return GreaterThanZero(childSize) && LessOrEqualThan(childSize, parentSize);
    }

    /// <summary>
    /// <para>
    /// Gets the balance value using the specified stored value.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <param name="storedValue">
    /// <para>The stored value.</para>
    /// <para></para>
    /// </param>
    /// <returns>
    /// <para>The sbyte</para>
    /// <para></para>
    /// </returns>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    protected virtual sbyte GetBalanceValue(TLinkAddress storedValue)
    {
        unchecked
        {
```

```
455                 var value =
                    ↪ _addressToInt32Converter.Convert(Bit<TLinkAddress>.PartialRead(storedValue,
                    ↪ 0, 3));
456                 value |= 0xF8 * ((value & 4) >> 2); // if negative, then continue ones to the
                    ↪ end of sbyte
457                 return (sbyte)value;
458             }
459         }
460
461         /// <summary>
462         /// <para>
463         /// Sets the balance value using the specified stored value.
464         /// </para>
465         /// <para></para>
466         /// </summary>
467         /// <param name="storedValue">
468         /// <para>The stored value.</para>
469         /// <para></para>
470         /// </param>
471         /// <param name="value">
472         /// <para>The value.</para>
473         /// <para></para>
474         /// </param>
475         [MethodImpl(MethodImplOptions.AggressiveInlining)]
476         protected virtual void SetBalanceValue(ref TLinkAddress storedValue, sbyte value)
477         {
478             unchecked
479             {
480                 var packagedValue = _int32ToAddressConverter.Convert((byte)value >> 5 & 4 |
                    ↪ value & 3);
481                 var modified = Bit<TLinkAddress>.PartialWrite(storedValue, packagedValue, 0, 3);
482                 storedValue = modified;
483             }
484         }
485
486         /// <summary>
487         /// <para>
488         /// The zero.
489         /// </para>
490         /// <para></para>
491         /// </summary>
492         public TLinkAddress this[TLinkAddress index]
493         {
494             [MethodImpl(MethodImplOptions.AggressiveInlining)]
495             get
496             {
497                 var root = GetTreeRoot();
498                 if (GreaterOrEqualThan(index, GetSize(root)))
499                 {
500                     return Zero;
501                 }
502                 while (!EqualToZero(root))
503                 {
504                     var left = GetLeftOrDefault(root);
505                     var leftSize = GetSizeOrZero(left);
506                     if (LessThan(index, leftSize))
507                     {
508                         root = left;
509                         continue;
510                     }
511                     if (AreEqual(index, leftSize))
512                     {
513                         return root;
514                     }
515                     root = GetRightOrDefault(root);
516                     index = Subtract(index, Increment(leftSize));
517                 }
518                 return Zero; // TODO: Impossible situation exception (only if tree structure
                    ↪ broken)
519             }
520         }
521
522         /// <summary>
523         /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
                ↪ (концом).
524         /// </summary>
525         /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
526         /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
```

```
527             /// <returns>Индекс искомой связи.</returns>
528             [MethodImpl(MethodImplOptions.AggressiveInlining)]
529             public TLinkAddress Search(TLinkAddress source, TLinkAddress target)
530             {
531                 var root = GetTreeRoot();
532                 while (!EqualToZero(root))
533                 {
534                     ref var rootLink = ref GetLinkReference(root);
535                     var rootSource = rootLink.Source;
536                     var rootTarget = rootLink.Target;
537                     if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                        ↪ node.Key < root.Key
538                     {
539                         root = GetLeftOrDefault(root);
540                     }
541                     else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
                        ↪ node.Key > root.Key
542                     {
543                         root = GetRightOrDefault(root);
544                     }
545                     else // node.Key == root.Key
546                     {
547                         return root;
548                     }
549                 }
550                 return Zero;
551             }
552
553             // TODO: Return indices range instead of references count
554             /// <summary>
555             /// <para>
556             /// Counts the usages using the specified link.
557             /// </para>
558             /// <para></para>
559             /// </summary>
560             /// <param name="link">
561             /// <para>The link.</para>
562             /// <para></para>
563             /// </param>
564             /// <returns>
565             /// <para>The link</para>
566             /// <para></para>
567             /// </returns>
568             [MethodImpl(MethodImplOptions.AggressiveInlining)]
569             public TLinkAddress CountUsages(TLinkAddress link)
570             {
571                 var root = GetTreeRoot();
572                 var total = GetSize(root);
573                 var totalRightIgnore = Zero;
574                 while (!EqualToZero(root))
575                 {
576                     var @base = GetBasePartValue(root);
577                     if (LessOrEqualThan(@base, link))
578                     {
579                         root = GetRightOrDefault(root);
580                     }
581                     else
582                     {
583                         totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
584                         root = GetLeftOrDefault(root);
585                     }
586                 }
587                 root = GetTreeRoot();
588                 var totalLeftIgnore = Zero;
589                 while (!EqualToZero(root))
590                 {
591                     var @base = GetBasePartValue(root);
592                     if (GreaterOrEqualThan(@base, link))
593                     {
594                         root = GetLeftOrDefault(root);
595                     }
596                     else
597                     {
598                         totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
599
600                         root = GetRightOrDefault(root);
601                     }
602                 }
```

```
603             return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
604         }
605 
606         /// <summary>
607         /// <para>
608         /// Eaches the usage using the specified link.
609         /// </para>
610         /// <para></para>
611         /// </summary>
612         /// <param name="link">
613         /// <para>The link.</para>
614         /// <para></para>
615         /// </param>
616         /// <param name="handler">
617         /// <para>The handler.</para>
618         /// <para></para>
619         /// </param>
620         /// <returns>
621         /// <para>The continue.</para>
622         /// <para></para>
623         /// </returns>
624         [MethodImpl(MethodImplOptions.AggressiveInlining)]
625         public TLinkAddress EachUsage(TLinkAddress link, ReadHandler<TLinkAddress>? handler)
626         {
627             var root = GetTreeRoot();
628             if (EqualToZero(root))
629             {
630                 return Continue;
631             }
632             TLinkAddress first = Zero, current = root;
633             while (!EqualToZero(current))
634             {
635                 var @base = GetBasePartValue(current);
636                 if (GreaterOrEqualThan(@base, link))
637                 {
638                     if (AreEqual(@base, link))
639                     {
640                         first = current;
641                     }
642                     current = GetLeftOrDefault(current);
643                 }
644                 else
645                 {
646                     current = GetRightOrDefault(current);
647                 }
648             }
649             if (!EqualToZero(first))
650             {
651                 current = first;
652                 while (true)
653                 {
654                     if (AreEqual(handler(GetLinkValues(current)), Break))
655                     {
656                         return Break;
657                     }
658                     current = GetNext(current);
659                     if (EqualToZero(current) || !AreEqual(GetBasePartValue(current), link))
660                     {
661                         break;
662                     }
663                 }
664             }
665             return Continue;
666         }
667 
668         /// <summary>
669         /// <para>
670         /// Prints the node value using the specified node.
671         /// </para>
672         /// <para></para>
673         /// </summary>
674         /// <param name="node">
675         /// <para>The node.</para>
676         /// <para></para>
677         /// </param>
678         /// <param name="sb">
679         /// <para>The sb.</para>
680         /// <para></para>
```

```
681         /// </param>
682         [MethodImpl(MethodImplOptions.AggressiveInlining)]
683         protected override void PrintNodeValue(TLinkAddress node, StringBuilder sb)
684         {
685             ref var link = ref GetLinkReference(node);
686             sb.Append(' ');
687             sb.Append(link.Source);
688             sb.Append('-');
689             sb.Append('>');
690             sb.Append(link.Target);
691         }
692     }
693 }
```

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

```
1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Converters;
7  using Platform.Delegates;
8  using static System.Runtime.CompilerServices.Unsafe;
9
10 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12 namespace Platform.Data.Doublets.Memory.United.Generic
13 {
14     /// <summary>
15     /// <para>
16     /// Represents the links recursionless size balanced tree methods base.
17     /// </para>
18     /// <para></para>
19     /// </summary>
20     /// <seealso cref="RecursionlessSizeBalancedTreeMethods{TLinkAddress}"/>
21     /// <seealso cref="ILinksTreeMethods{TLinkAddress}"/>
22     public unsafe abstract class LinksRecursionlessSizeBalancedTreeMethodsBase<TLinkAddress> :
       ↪ RecursionlessSizeBalancedTreeMethods<TLinkAddress>, ILinksTreeMethods<TLinkAddress>
23     {
24         private static readonly UncheckedConverter<TLinkAddress, long> _addressToInt64Converter
           ↪ = UncheckedConverter<TLinkAddress, long>.Default;
25
26         /// <summary>
27         /// <para>
28         /// The break.
29         /// </para>
30         /// <para></para>
31         /// </summary>
32         protected readonly TLinkAddress Break;
33         /// <summary>
34         /// <para>
35         /// The continue.
36         /// </para>
37         /// <para></para>
38         /// </summary>
39         protected readonly TLinkAddress Continue;
40         /// <summary>
41         /// <para>
42         /// The links.
43         /// </para>
44         /// <para></para>
45         /// </summary>
46         protected readonly byte* Links;
47         /// <summary>
48         /// <para>
49         /// The header.
50         /// </para>
51         /// <para></para>
52         /// </summary>
53         protected readonly byte* Header;
54
55         /// <summary>
56         /// <para>
57         /// Initializes a new <see cref="LinksRecursionlessSizeBalancedTreeMethodsBase"/>
           ↪ instance.
58         /// </para>
59         /// <para></para>
60         /// </summary>
61         /// <param name="constants">
```

```
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected LinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLinkAddress>
        ↪  constants, byte* links, byte* header)
        {
            Links = links;
            Header = header;
            Break = constants.Break;
            Continue = constants.Continue;
        }

        /// <summary>
        /// <para>
        /// Gets the tree root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TLinkAddress GetTreeRoot();

        /// <summary>
        /// <para>
        /// Gets the base part value using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TLinkAddress GetBasePartValue(TLinkAddress link);

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the right of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <param name="rootSource">
        /// <para>The root source.</para>
        /// <para></para>
        /// </param>
        /// <param name="rootTarget">
        /// <para>The root target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        protected abstract bool FirstIsToTheRightOfSecond(TLinkAddress source, TLinkAddress target, TLinkAddress rootSource, TLinkAddress rootTarget);

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <param name="rootSource">
        /// <para>The root source.</para>
        /// <para></para>
        /// </param>
        /// <param name="rootTarget">
        /// <para>The root target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract bool FirstIsToTheLeftOfSecond(TLinkAddress source, TLinkAddress target, TLinkAddress rootSource, TLinkAddress rootTarget);

        /// <summary>
        /// <para>
        /// Gets the header reference.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A ref links header of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref LinksHeader<TLinkAddress> GetHeaderReference() => ref AsRef<LinksHeader<TLinkAddress>>(Header);

        /// <summary>
        /// <para>
        /// Gets the link reference using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref RawLink<TLinkAddress> GetLinkReference(TLinkAddress link) => ref AsRef<RawLink<TLinkAddress>>(Links + (RawLink<TLinkAddress>.SizeInBytes * _addressToInt64Converter.Convert(link)));

        /// <summary>
        /// <para>
        /// Gets the link values using the specified link index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="linkIndex">
        /// <para>The link index.</para>
        /// <para></para>
        /// </param>
        /// <returns>
```

```
        /// <para>A list of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual IList<TLinkAddress>? GetLinkValues(TLinkAddress linkIndex)
        {
            ref var link = ref GetLinkReference(linkIndex);
            return new Link<TLinkAddress>(linkIndex, link.Source, link.Target);
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLinkAddress first, TLinkAddress second)
        {
            ref var firstLink = ref GetLinkReference(first);
            ref var secondLink = ref GetLinkReference(second);
            return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
             ↪  secondLink.Source, secondLink.Target);
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the right of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLinkAddress first, TLinkAddress
         ↪  second)
        {
            ref var firstLink = ref GetLinkReference(first);
            ref var secondLink = ref GetLinkReference(second);
            return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
             ↪  secondLink.Source, secondLink.Target);
        }

        /// <summary>
        /// <para>
        /// The zero.
        /// </para>
        /// <para></para>
        /// </summary>
        public TLinkAddress this[TLinkAddress index]
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get
            {
                var root = GetTreeRoot();
                if (GreaterOrEqualThan(index, GetSize(root)))
```

```
286                 {
287                     return Zero;
288                 }
289                 while (!EqualToZero(root))
290                 {
291                     var left = GetLeftOrDefault(root);
292                     var leftSize = GetSizeOrZero(left);
293                     if (LessThan(index, leftSize))
294                     {
295                         root = left;
296                         continue;
297                     }
298                     if (AreEqual(index, leftSize))
299                     {
300                         return root;
301                     }
302                     root = GetRightOrDefault(root);
303                     index = Subtract(index, Increment(leftSize));
304                 }
305                 return Zero; // TODO: Impossible situation exception (only if tree structure
                    ↪ broken)
306             }
307         }
308 
309         /// <summary>
310         /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
                ↪ (концом).
311         /// </summary>
312         /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
313         /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
314         /// <returns>Индекс искомой связи.</returns>
315         [MethodImpl(MethodImplOptions.AggressiveInlining)]
316         public TLinkAddress Search(TLinkAddress source, TLinkAddress target)
317         {
318             var root = GetTreeRoot();
319             while (!EqualToZero(root))
320             {
321                 ref var rootLink = ref GetLinkReference(root);
322                 var rootSource = rootLink.Source;
323                 var rootTarget = rootLink.Target;
324                 if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                    ↪ node.Key < root.Key
325                 {
326                     root = GetLeftOrDefault(root);
327                 }
328                 else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
                    ↪ node.Key > root.Key
329                 {
330                     root = GetRightOrDefault(root);
331                 }
332                 else // node.Key == root.Key
333                 {
334                     return root;
335                 }
336             }
337             return Zero;
338         }
339 
340         // TODO: Return indices range instead of references count
341         /// <summary>
342         /// <para>
343         /// Counts the usages using the specified link.
344         /// </para>
345         /// <para></para>
346         /// </summary>
347         /// <param name="link">
348         /// <para>The link.</para>
349         /// <para></para>
350         /// </param>
351         /// <returns>
352         /// <para>The link</para>
353         /// <para></para>
354         /// </returns>
355         [MethodImpl(MethodImplOptions.AggressiveInlining)]
356         public TLinkAddress CountUsages(TLinkAddress link)
357         {
358             var root = GetTreeRoot();
359             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);
        }

        /// <summary>
        /// <para>
        /// Eaches the usage using the specified base.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="@base">
        /// <para>The base.</para>
        /// <para></para>
        /// </param>
        /// <param name="handler">
        /// <para>The handler.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress EachUsage(TLinkAddress @base, ReadHandler<TLinkAddress>? 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 TLinkAddress EachUsageCore(TLinkAddress @base, TLinkAddress link,
        ↪  ReadHandler<TLinkAddress>? 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;
        }

        /// <summary>
        /// <para>
        /// Prints the node value using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="sb">
        /// <para>The sb.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void PrintNodeValue(TLinkAddress node, StringBuilder sb)
        {
            ref var link = ref GetLinkReference(node);
            sb.Append(' ');
            sb.Append(link.Source);
            sb.Append('-');
            sb.Append('>');
            sb.Append(link.Target);
        }
    }
}
```

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

```
using System;
using System.Text;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections.Methods.Trees;
using Platform.Converters;
using Platform.Delegates;
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
{
    /// <summary>
    /// <para>
    /// Represents the links size balanced tree methods base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="SizeBalancedTreeMethods{TLinkAddress}"/>
    /// <seealso cref="ILinksTreeMethods{TLinkAddress}"/>
    public unsafe abstract class LinksSizeBalancedTreeMethodsBase<TLinkAddress> :
    ↪  SizeBalancedTreeMethods<TLinkAddress>, ILinksTreeMethods<TLinkAddress>
    {
        private static readonly UncheckedConverter<TLinkAddress, long> _addressToInt64Converter
        ↪  = UncheckedConverter<TLinkAddress, long>.Default;

        /// <summary>
        /// <para>
        /// The break.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly TLinkAddress Break;
        /// <summary>
        /// <para>
        /// The continue.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly TLinkAddress Continue;
        /// <summary>
        /// <para>
        /// The links.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly byte* Links;
        /// <summary>
        /// <para>
        /// The header.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly byte* Header;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksSizeBalancedTreeMethodsBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected LinksSizeBalancedTreeMethodsBase(LinksConstants<TLinkAddress> constants, byte*
        ↪  links, byte* header)
        {
            Links = links;
            Header = header;
            Break = constants.Break;
            Continue = constants.Continue;
        }

        /// <summary>
        /// <para>
        /// Gets the tree root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TLinkAddress GetTreeRoot();

        /// <summary>
        /// <para>
        /// Gets the base part value using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
```

```
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TLinkAddress GetBasePartValue(TLinkAddress link);

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the right of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <param name="rootSource">
        /// <para>The root source.</para>
        /// <para></para>
        /// </param>
        /// <param name="rootTarget">
        /// <para>The root target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract bool FirstIsToTheRightOfSecond(TLinkAddress source, TLinkAddress
        ↪  target, TLinkAddress rootSource, TLinkAddress rootTarget);

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <param name="rootSource">
        /// <para>The root source.</para>
        /// <para></para>
        /// </param>
        /// <param name="rootTarget">
        /// <para>The root target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract bool FirstIsToTheLeftOfSecond(TLinkAddress source, TLinkAddress
        ↪  target, TLinkAddress rootSource, TLinkAddress rootTarget);

        /// <summary>
        /// <para>
        /// Gets the header reference.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A ref links header of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
181         protected virtual ref LinksHeader<TLinkAddress> GetHeaderReference() => ref
            ↪ AsRef<LinksHeader<TLinkAddress>>(Header);
182
183         /// <summary>
184         /// <para>
185         /// Gets the link reference using the specified link.
186         /// </para>
187         /// <para></para>
188         /// </summary>
189         /// <param name="link">
190         /// <para>The link.</para>
191         /// <para></para>
192         /// </param>
193         /// <returns>
194         /// <para>A ref raw link of t link</para>
195         /// <para></para>
196         /// </returns>
197         [MethodImpl(MethodImplOptions.AggressiveInlining)]
198         protected virtual ref RawLink<TLinkAddress> GetLinkReference(TLinkAddress link) => ref
            ↪ AsRef<RawLink<TLinkAddress>>(Links + (RawLink<TLinkAddress>.SizeInBytes *
            ↪ _addressToInt64Converter.Convert(link)));
199
200         /// <summary>
201         /// <para>
202         /// Gets the link values using the specified link index.
203         /// </para>
204         /// <para></para>
205         /// </summary>
206         /// <param name="linkIndex">
207         /// <para>The link index.</para>
208         /// <para></para>
209         /// </param>
210         /// <returns>
211         /// <para>A list of t link</para>
212         /// <para></para>
213         /// </returns>
214         [MethodImpl(MethodImplOptions.AggressiveInlining)]
215         protected virtual IList<TLinkAddress>? GetLinkValues(TLinkAddress linkIndex)
216         {
217             ref var link = ref GetLinkReference(linkIndex);
218             return new Link<TLinkAddress>(linkIndex, link.Source, link.Target);
219         }
220
221         /// <summary>
222         /// <para>
223         /// Determines whether this instance first is to the left of second.
224         /// </para>
225         /// <para></para>
226         /// </summary>
227         /// <param name="first">
228         /// <para>The first.</para>
229         /// <para></para>
230         /// </param>
231         /// <param name="second">
232         /// <para>The second.</para>
233         /// <para></para>
234         /// </param>
235         /// <returns>
236         /// <para>The bool</para>
237         /// <para></para>
238         /// </returns>
239         [MethodImpl(MethodImplOptions.AggressiveInlining)]
240         protected override bool FirstIsToTheLeftOfSecond(TLinkAddress first, TLinkAddress second)
241         {
242             ref var firstLink = ref GetLinkReference(first);
243             ref var secondLink = ref GetLinkReference(second);
244             return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
                ↪ secondLink.Source, secondLink.Target);
245         }
246
247         /// <summary>
248         /// <para>
249         /// Determines whether this instance first is to the right of second.
250         /// </para>
251         /// <para></para>
252         /// </summary>
253         /// <param name="first">
254         /// <para>The first.</para>
```

```
255             /// <para></para>
256             /// </param>
257             /// <param name="second">
258             /// <para>The second.</para>
259             /// <para></para>
260             /// </param>
261             /// <returns>
262             /// <para>The bool</para>
263             /// <para></para>
264             /// </returns>
265             [MethodImpl(MethodImplOptions.AggressiveInlining)]
266             protected override bool FirstIsToTheRightOfSecond(TLinkAddress first, TLinkAddress
                ↪ second)
267             {
268                 ref var firstLink = ref GetLinkReference(first);
269                 ref var secondLink = ref GetLinkReference(second);
270                 return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
                    ↪ secondLink.Source, secondLink.Target);
271             }
272
273             /// <summary>
274             /// <para>
275             /// The zero.
276             /// </para>
277             /// <para></para>
278             /// </summary>
279             public TLinkAddress this[TLinkAddress index]
280             {
281                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
282                 get
283                 {
284                     var root = GetTreeRoot();
285                     if (GreaterOrEqualThan(index, GetSize(root)))
286                     {
287                         return Zero;
288                     }
289                     while (!EqualToZero(root))
290                     {
291                         var left = GetLeftOrDefault(root);
292                         var leftSize = GetSizeOrZero(left);
293                         if (LessThan(index, leftSize))
294                         {
295                             root = left;
296                             continue;
297                         }
298                         if (AreEqual(index, leftSize))
299                         {
300                             return root;
301                         }
302                         root = GetRightOrDefault(root);
303                         index = Subtract(index, Increment(leftSize));
304                     }
305                     return Zero; // TODO: Impossible situation exception (only if tree structure
                        ↪ broken)
306                 }
307             }
308
309             /// <summary>
310             /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
                ↪ (концом).
311             /// </summary>
312             /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
313             /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
314             /// <returns>Индекс искомой связи.</returns>
315             [MethodImpl(MethodImplOptions.AggressiveInlining)]
316             public TLinkAddress Search(TLinkAddress source, TLinkAddress target)
317             {
318                 var root = GetTreeRoot();
319                 while (!EqualToZero(root))
320                 {
321                     ref var rootLink = ref GetLinkReference(root);
322                     var rootSource = rootLink.Source;
323                     var rootTarget = rootLink.Target;
324                     if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                        ↪ node.Key < root.Key
325                     {
326                         root = GetLeftOrDefault(root);
327                     }
```

```
                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
        /// <summary>
        /// <para>
        /// Counts the usages using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress CountUsages(TLinkAddress 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);
        }

        /// <summary>
        /// <para>
        /// Eaches the usage using the specified base.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="@base">
        /// <para>The base.</para>
        /// <para></para>
        /// </param>
        /// <param name="handler">
        /// <para>The handler.</para>
        /// <para></para>
```

```
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress EachUsage(TLinkAddress @base, ReadHandler<TLinkAddress>? 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 TLinkAddress EachUsageCore(TLinkAddress @base, TLinkAddress link,
        ↪  ReadHandler<TLinkAddress>? 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;
        }

        /// <summary>
        /// <para>
        /// Prints the node value using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="sb">
        /// <para>The sb.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void PrintNodeValue(TLinkAddress node, StringBuilder sb)
        {
            ref var link = ref GetLinkReference(node);
            sb.Append(' ');
            sb.Append(link.Source);
            sb.Append('-');
            sb.Append('>');
            sb.Append(link.Target);
        }
```

```
    }
}
```

1.83 ./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
{
    /// <summary>
    /// <para>
    /// Represents the links sources avl balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksAvlBalancedTreeMethodsBase{TLinkAddress}"/>
    public unsafe class LinksSourcesAvlBalancedTreeMethods<TLinkAddress> :
    ↪ LinksAvlBalancedTreeMethodsBase<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksSourcesAvlBalancedTreeMethods"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksSourcesAvlBalancedTreeMethods(LinksConstants<TLinkAddress> constants, byte*
        ↪ links, byte* header) : base(constants, links, header) { }

        /// <summary>
        /// <para>
        /// Gets the left reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
        ↪ GetLinkReference(node).LeftAsSource;

        /// <summary>
        /// <para>
        /// Gets the right reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
        ↪ GetLinkReference(node).RightAsSource;

```

```
71          /// <summary>
72          /// <para>
73          /// Gets the left using the specified node.
74          /// </para>
75          /// <para></para>
76          /// </summary>
77          /// <param name="node">
78          /// <para>The node.</para>
79          /// <para></para>
80          /// </param>
81          /// <returns>
82          /// <para>The link</para>
83          /// <para></para>
84          /// </returns>
85          [MethodImpl(MethodImplOptions.AggressiveInlining)]
86          protected override TLinkAddress GetLeft(TLinkAddress node) =>
            ↪ GetLinkReference(node).LeftAsSource;
87
88          /// <summary>
89          /// <para>
90          /// Gets the right using the specified node.
91          /// </para>
92          /// <para></para>
93          /// </summary>
94          /// <param name="node">
95          /// <para>The node.</para>
96          /// <para></para>
97          /// </param>
98          /// <returns>
99          /// <para>The link</para>
100         /// <para></para>
101         /// </returns>
102         [MethodImpl(MethodImplOptions.AggressiveInlining)]
103         protected override TLinkAddress GetRight(TLinkAddress node) =>
            ↪ GetLinkReference(node).RightAsSource;
104
105         /// <summary>
106         /// <para>
107         /// Sets the left using the specified node.
108         /// </para>
109         /// <para></para>
110         /// </summary>
111         /// <param name="node">
112         /// <para>The node.</para>
113         /// <para></para>
114         /// </param>
115         /// <param name="left">
116         /// <para>The left.</para>
117         /// <para></para>
118         /// </param>
119         [MethodImpl(MethodImplOptions.AggressiveInlining)]
120         protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
            ↪ GetLinkReference(node).LeftAsSource = left;
121
122         /// <summary>
123         /// <para>
124         /// Sets the right using the specified node.
125         /// </para>
126         /// <para></para>
127         /// </summary>
128         /// <param name="node">
129         /// <para>The node.</para>
130         /// <para></para>
131         /// </param>
132         /// <param name="right">
133         /// <para>The right.</para>
134         /// <para></para>
135         /// </param>
136         [MethodImpl(MethodImplOptions.AggressiveInlining)]
137         protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
            ↪ GetLinkReference(node).RightAsSource = right;
138
139         /// <summary>
140         /// <para>
141         /// Gets the size using the specified node.
142         /// </para>
143         /// <para></para>
144         /// </summary>
```

```
145         /// <param name="node">
146         /// <para>The node.</para>
147         /// <para></para>
148         /// </param>
149         /// <returns>
150         /// <para>The link</para>
151         /// <para></para>
152         /// </returns>
153         [MethodImpl(MethodImplOptions.AggressiveInlining)]
154         protected override TLinkAddress GetSize(TLinkAddress node) =>
            ↪ GetSizeValue(GetLinkReference(node).SizeAsSource);
155
156         /// <summary>
157         /// <para>
158         /// Sets the size using the specified node.
159         /// </para>
160         /// <para></para>
161         /// </summary>
162         /// <param name="node">
163         /// <para>The node.</para>
164         /// <para></para>
165         /// </param>
166         /// <param name="size">
167         /// <para>The size.</para>
168         /// <para></para>
169         /// </param>
170         [MethodImpl(MethodImplOptions.AggressiveInlining)]
171         protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
            ↪ SetSizeValue(ref GetLinkReference(node).SizeAsSource, size);
172
173         /// <summary>
174         /// <para>
175         /// Determines whether this instance get left is child.
176         /// </para>
177         /// <para></para>
178         /// </summary>
179         /// <param name="node">
180         /// <para>The node.</para>
181         /// <para></para>
182         /// </param>
183         /// <returns>
184         /// <para>The bool</para>
185         /// <para></para>
186         /// </returns>
187         [MethodImpl(MethodImplOptions.AggressiveInlining)]
188         protected override bool GetLeftIsChild(TLinkAddress node) =>
            ↪ GetLeftIsChildValue(GetLinkReference(node).SizeAsSource);
189
190         /// <summary>
191         /// <para>
192         /// Sets the left is child using the specified node.
193         /// </para>
194         /// <para></para>
195         /// </summary>
196         /// <param name="node">
197         /// <para>The node.</para>
198         /// <para></para>
199         /// </param>
200         /// <param name="value">
201         /// <para>The value.</para>
202         /// <para></para>
203         /// </param>
204         [MethodImpl(MethodImplOptions.AggressiveInlining)]
205         protected override void SetLeftIsChild(TLinkAddress node, bool value) =>
            ↪ SetLeftIsChildValue(ref GetLinkReference(node).SizeAsSource, value);
206
207         /// <summary>
208         /// <para>
209         /// Determines whether this instance get right is child.
210         /// </para>
211         /// <para></para>
212         /// </summary>
213         /// <param name="node">
214         /// <para>The node.</para>
215         /// <para></para>
216         /// </param>
217         /// <returns>
218         /// <para>The bool</para>
```

```
219         /// <para></para>
220         /// </returns>
221         [MethodImpl(MethodImplOptions.AggressiveInlining)]
222         protected override bool GetRightIsChild(TLinkAddress node) =>
            ↪ GetRightIsChildValue(GetLinkReference(node).SizeAsSource);
223
224         /// <summary>
225         /// <para>
226         /// Sets the right is child using the specified node.
227         /// </para>
228         /// <para></para>
229         /// </summary>
230         /// <param name="node">
231         /// <para>The node.</para>
232         /// <para></para>
233         /// </param>
234         /// <param name="value">
235         /// <para>The value.</para>
236         /// <para></para>
237         /// </param>
238         [MethodImpl(MethodImplOptions.AggressiveInlining)]
239         protected override void SetRightIsChild(TLinkAddress node, bool value) =>
            ↪ SetRightIsChildValue(ref GetLinkReference(node).SizeAsSource, value);
240
241         /// <summary>
242         /// <para>
243         /// Gets the balance using the specified node.
244         /// </para>
245         /// <para></para>
246         /// </summary>
247         /// <param name="node">
248         /// <para>The node.</para>
249         /// <para></para>
250         /// </param>
251         /// <returns>
252         /// <para>The sbyte</para>
253         /// <para></para>
254         /// </returns>
255         [MethodImpl(MethodImplOptions.AggressiveInlining)]
256         protected override sbyte GetBalance(TLinkAddress node) =>
            ↪ GetBalanceValue(GetLinkReference(node).SizeAsSource);
257
258         /// <summary>
259         /// <para>
260         /// Sets the balance using the specified node.
261         /// </para>
262         /// <para></para>
263         /// </summary>
264         /// <param name="node">
265         /// <para>The node.</para>
266         /// <para></para>
267         /// </param>
268         /// <param name="value">
269         /// <para>The value.</para>
270         /// <para></para>
271         /// </param>
272         [MethodImpl(MethodImplOptions.AggressiveInlining)]
273         protected override void SetBalance(TLinkAddress node, sbyte value) =>
            ↪ SetBalanceValue(ref GetLinkReference(node).SizeAsSource, value);
274
275         /// <summary>
276         /// <para>
277         /// Gets the tree root.
278         /// </para>
279         /// <para></para>
280         /// </summary>
281         /// <returns>
282         /// <para>The link</para>
283         /// <para></para>
284         /// </returns>
285         [MethodImpl(MethodImplOptions.AggressiveInlining)]
286         protected override TLinkAddress GetTreeRoot() => GetHeaderReference().RootAsSource;
287
288         /// <summary>
289         /// <para>
290         /// Gets the base part value using the specified link.
291         /// </para>
292         /// <para></para>
```

```
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetBasePartValue(TLinkAddress link) =>
        ↪  GetLinkReference(link).Source;

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="firstSource">
        /// <para>The first source.</para>
        /// <para></para>
        /// </param>
        /// <param name="firstTarget">
        /// <para>The first target.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondSource">
        /// <para>The second source.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondTarget">
        /// <para>The second target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLinkAddress firstSource, TLinkAddress
        ↪  firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget) =>
        ↪  LessThan(firstSource, secondSource) || (AreEqual(firstSource, secondSource) &&
        ↪  LessThan(firstTarget, secondTarget));

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the right of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="firstSource">
        /// <para>The first source.</para>
        /// <para></para>
        /// </param>
        /// <param name="firstTarget">
        /// <para>The first target.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondSource">
        /// <para>The second source.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondTarget">
        /// <para>The second target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLinkAddress firstSource, TLinkAddress
        ↪  firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget) =>
        ↪  GreaterThan(firstSource, secondSource) || (AreEqual(firstSource, secondSource) &&
        ↪  GreaterThan(firstTarget, secondTarget));

```

```
        /// <summary>
        /// <para>
        /// Clears the node using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLinkAddress node)
        {
            ref var link = ref GetLinkReference(node);
            link.LeftAsSource = Zero;
            link.RightAsSource = Zero;
            link.SizeAsSource = Zero;
        }
    }
}
```

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

```
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Memory.United.Generic
{
    /// <summary>
    /// <para>
    /// Represents the links sources recursionless size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksRecursionlessSizeBalancedTreeMethodsBase{TLinkAddress}"/>
    public unsafe class LinksSourcesRecursionlessSizeBalancedTreeMethods<TLinkAddress> :
    ↪ LinksRecursionlessSizeBalancedTreeMethodsBase<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksSourcesRecursionlessSizeBalancedTreeMethods"/>
        ↪ instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<TLinkAddress>
        ↪ constants, byte* links, byte* header) : base(constants, links, header) { }

        /// <summary>
        /// <para>
        /// Gets the left reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
        ↪ GetLinkReference(node).LeftAsSource;
```

```
53
54          /// <summary>
55          /// <para>
56          /// Gets the right reference using the specified node.
57          /// </para>
58          /// <para></para>
59          /// </summary>
60          /// <param name="node">
61          /// <para>The node.</para>
62          /// <para></para>
63          /// </param>
64          /// <returns>
65          /// <para>The ref link</para>
66          /// <para></para>
67          /// </returns>
68          [MethodImpl(MethodImplOptions.AggressiveInlining)]
69          protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
            ↪ GetLinkReference(node).RightAsSource;
70
71          /// <summary>
72          /// <para>
73          /// Gets the left using the specified node.
74          /// </para>
75          /// <para></para>
76          /// </summary>
77          /// <param name="node">
78          /// <para>The node.</para>
79          /// <para></para>
80          /// </param>
81          /// <returns>
82          /// <para>The link</para>
83          /// <para></para>
84          /// </returns>
85          [MethodImpl(MethodImplOptions.AggressiveInlining)]
86          protected override TLinkAddress GetLeft(TLinkAddress node) =>
            ↪ GetLinkReference(node).LeftAsSource;
87
88          /// <summary>
89          /// <para>
90          /// Gets the right using the specified node.
91          /// </para>
92          /// <para></para>
93          /// </summary>
94          /// <param name="node">
95          /// <para>The node.</para>
96          /// <para></para>
97          /// </param>
98          /// <returns>
99          /// <para>The link</para>
100         /// <para></para>
101         /// </returns>
102         [MethodImpl(MethodImplOptions.AggressiveInlining)]
103         protected override TLinkAddress GetRight(TLinkAddress node) =>
            ↪ GetLinkReference(node).RightAsSource;
104
105         /// <summary>
106         /// <para>
107         /// Sets the left using the specified node.
108         /// </para>
109         /// <para></para>
110         /// </summary>
111         /// <param name="node">
112         /// <para>The node.</para>
113         /// <para></para>
114         /// </param>
115         /// <param name="left">
116         /// <para>The left.</para>
117         /// <para></para>
118         /// </param>
119         [MethodImpl(MethodImplOptions.AggressiveInlining)]
120         protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
            ↪ GetLinkReference(node).LeftAsSource = left;
121
122         /// <summary>
123         /// <para>
124         /// Sets the right using the specified node.
125         /// </para>
126         /// <para></para>
```

```
127             /// </summary>
128             /// <param name="node">
129             /// <para>The node.</para>
130             /// <para></para>
131             /// </param>
132             /// <param name="right">
133             /// <para>The right.</para>
134             /// <para></para>
135             /// </param>
136             [MethodImpl(MethodImplOptions.AggressiveInlining)]
137             protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
                ↪ GetLinkReference(node).RightAsSource = right;
138
139             /// <summary>
140             /// <para>
141             /// Gets the size using the specified node.
142             /// </para>
143             /// <para></para>
144             /// </summary>
145             /// <param name="node">
146             /// <para>The node.</para>
147             /// <para></para>
148             /// </param>
149             /// <returns>
150             /// <para>The link</para>
151             /// <para></para>
152             /// </returns>
153             [MethodImpl(MethodImplOptions.AggressiveInlining)]
154             protected override TLinkAddress GetSize(TLinkAddress node) =>
                ↪ GetLinkReference(node).SizeAsSource;
155
156             /// <summary>
157             /// <para>
158             /// Sets the size using the specified node.
159             /// </para>
160             /// <para></para>
161             /// </summary>
162             /// <param name="node">
163             /// <para>The node.</para>
164             /// <para></para>
165             /// </param>
166             /// <param name="size">
167             /// <para>The size.</para>
168             /// <para></para>
169             /// </param>
170             [MethodImpl(MethodImplOptions.AggressiveInlining)]
171             protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
                ↪ GetLinkReference(node).SizeAsSource = size;
172
173             /// <summary>
174             /// <para>
175             /// Gets the tree root.
176             /// </para>
177             /// <para></para>
178             /// </summary>
179             /// <returns>
180             /// <para>The link</para>
181             /// <para></para>
182             /// </returns>
183             [MethodImpl(MethodImplOptions.AggressiveInlining)]
184             protected override TLinkAddress GetTreeRoot() => GetHeaderReference().RootAsSource;
185
186             /// <summary>
187             /// <para>
188             /// Gets the base part value using the specified link.
189             /// </para>
190             /// <para></para>
191             /// </summary>
192             /// <param name="link">
193             /// <para>The link.</para>
194             /// <para></para>
195             /// </param>
196             /// <returns>
197             /// <para>The link</para>
198             /// <para></para>
199             /// </returns>
200             [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
201         protected override TLinkAddress GetBasePartValue(TLinkAddress link) =>
            ↪ GetLinkReference(link).Source;
202
203         /// <summary>
204         /// <para>
205         /// Determines whether this instance first is to the left of second.
206         /// </para>
207         /// <para></para>
208         /// </summary>
209         /// <param name="firstSource">
210         /// <para>The first source.</para>
211         /// <para></para>
212         /// </param>
213         /// <param name="firstTarget">
214         /// <para>The first target.</para>
215         /// <para></para>
216         /// </param>
217         /// <param name="secondSource">
218         /// <para>The second source.</para>
219         /// <para></para>
220         /// </param>
221         /// <param name="secondTarget">
222         /// <para>The second target.</para>
223         /// <para></para>
224         /// </param>
225         /// <returns>
226         /// <para>The bool</para>
227         /// <para></para>
228         /// </returns>
229         [MethodImpl(MethodImplOptions.AggressiveInlining)]
230         protected override bool FirstIsToTheLeftOfSecond(TLinkAddress firstSource, TLinkAddress
            ↪ firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget) =>
            ↪ LessThan(firstSource, secondSource) || (AreEqual(firstSource, secondSource) &&
            ↪ LessThan(firstTarget, secondTarget));
231
232         /// <summary>
233         /// <para>
234         /// Determines whether this instance first is to the right of second.
235         /// </para>
236         /// <para></para>
237         /// </summary>
238         /// <param name="firstSource">
239         /// <para>The first source.</para>
240         /// <para></para>
241         /// </param>
242         /// <param name="firstTarget">
243         /// <para>The first target.</para>
244         /// <para></para>
245         /// </param>
246         /// <param name="secondSource">
247         /// <para>The second source.</para>
248         /// <para></para>
249         /// </param>
250         /// <param name="secondTarget">
251         /// <para>The second target.</para>
252         /// <para></para>
253         /// </param>
254         /// <returns>
255         /// <para>The bool</para>
256         /// <para></para>
257         /// </returns>
258         [MethodImpl(MethodImplOptions.AggressiveInlining)]
259         protected override bool FirstIsToTheRightOfSecond(TLinkAddress firstSource, TLinkAddress
            ↪ firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget) =>
            ↪ GreaterThan(firstSource, secondSource) || (AreEqual(firstSource, secondSource) &&
            ↪ GreaterThan(firstTarget, secondTarget));
260
261         /// <summary>
262         /// <para>
263         /// Clears the node using the specified node.
264         /// </para>
265         /// <para></para>
266         /// </summary>
267         /// <param name="node">
268         /// <para>The node.</para>
269         /// <para></para>
270         /// </param>
```

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

1.85 ./csharp/Platform.Data.Doublets/Memory/United/Generic/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
{
    /// <summary>
    /// <para>
    /// Represents the links sources size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksSizeBalancedTreeMethodsBase{TLinkAddress}"/>
    public unsafe class LinksSourcesSizeBalancedTreeMethods<TLinkAddress> :
    ↪ LinksSizeBalancedTreeMethodsBase<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksSourcesSizeBalancedTreeMethods"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksSourcesSizeBalancedTreeMethods(LinksConstants<TLinkAddress> constants, byte*
        ↪ links, byte* header) : base(constants, links, header) { }

        /// <summary>
        /// <para>
        /// Gets the left reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
        ↪ GetLinkReference(node).LeftAsSource;

        /// <summary>
        /// <para>
        /// Gets the right reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
```

```
64          /// <returns>
65          /// <para>The ref link</para>
66          /// <para></para>
67          /// </returns>
68          [MethodImpl(MethodImplOptions.AggressiveInlining)]
69          protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
            ↪ GetLinkReference(node).RightAsSource;
70
71          /// <summary>
72          /// <para>
73          /// Gets the left using the specified node.
74          /// </para>
75          /// <para></para>
76          /// </summary>
77          /// <param name="node">
78          /// <para>The node.</para>
79          /// <para></para>
80          /// </param>
81          /// <returns>
82          /// <para>The link</para>
83          /// <para></para>
84          /// </returns>
85          [MethodImpl(MethodImplOptions.AggressiveInlining)]
86          protected override TLinkAddress GetLeft(TLinkAddress node) =>
            ↪ GetLinkReference(node).LeftAsSource;
87
88          /// <summary>
89          /// <para>
90          /// Gets the right using the specified node.
91          /// </para>
92          /// <para></para>
93          /// </summary>
94          /// <param name="node">
95          /// <para>The node.</para>
96          /// <para></para>
97          /// </param>
98          /// <returns>
99          /// <para>The link</para>
100         /// <para></para>
101         /// </returns>
102         [MethodImpl(MethodImplOptions.AggressiveInlining)]
103         protected override TLinkAddress GetRight(TLinkAddress node) =>
            ↪ GetLinkReference(node).RightAsSource;
104
105         /// <summary>
106         /// <para>
107         /// Sets the left using the specified node.
108         /// </para>
109         /// <para></para>
110         /// </summary>
111         /// <param name="node">
112         /// <para>The node.</para>
113         /// <para></para>
114         /// </param>
115         /// <param name="left">
116         /// <para>The left.</para>
117         /// <para></para>
118         /// </param>
119         [MethodImpl(MethodImplOptions.AggressiveInlining)]
120         protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
            ↪ GetLinkReference(node).LeftAsSource = left;
121
122         /// <summary>
123         /// <para>
124         /// Sets the right using the specified node.
125         /// </para>
126         /// <para></para>
127         /// </summary>
128         /// <param name="node">
129         /// <para>The node.</para>
130         /// <para></para>
131         /// </param>
132         /// <param name="right">
133         /// <para>The right.</para>
134         /// <para></para>
135         /// </param>
136         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
137         protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
            ↪ GetLinkReference(node).RightAsSource = right;
138
139         /// <summary>
140         /// <para>
141         /// Gets the size using the specified node.
142         /// </para>
143         /// <para></para>
144         /// </summary>
145         /// <param name="node">
146         /// <para>The node.</para>
147         /// <para></para>
148         /// </param>
149         /// <returns>
150         /// <para>The link</para>
151         /// <para></para>
152         /// </returns>
153         [MethodImpl(MethodImplOptions.AggressiveInlining)]
154         protected override TLinkAddress GetSize(TLinkAddress node) =>
            ↪ GetLinkReference(node).SizeAsSource;
155
156         /// <summary>
157         /// <para>
158         /// Sets the size using the specified node.
159         /// </para>
160         /// <para></para>
161         /// </summary>
162         /// <param name="node">
163         /// <para>The node.</para>
164         /// <para></para>
165         /// </param>
166         /// <param name="size">
167         /// <para>The size.</para>
168         /// <para></para>
169         /// </param>
170         [MethodImpl(MethodImplOptions.AggressiveInlining)]
171         protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
            ↪ GetLinkReference(node).SizeAsSource = size;
172
173         /// <summary>
174         /// <para>
175         /// Gets the tree root.
176         /// </para>
177         /// <para></para>
178         /// </summary>
179         /// <returns>
180         /// <para>The link</para>
181         /// <para></para>
182         /// </returns>
183         [MethodImpl(MethodImplOptions.AggressiveInlining)]
184         protected override TLinkAddress GetTreeRoot() => GetHeaderReference().RootAsSource;
185
186         /// <summary>
187         /// <para>
188         /// Gets the base part value using the specified link.
189         /// </para>
190         /// <para></para>
191         /// </summary>
192         /// <param name="link">
193         /// <para>The link.</para>
194         /// <para></para>
195         /// </param>
196         /// <returns>
197         /// <para>The link</para>
198         /// <para></para>
199         /// </returns>
200         [MethodImpl(MethodImplOptions.AggressiveInlining)]
201         protected override TLinkAddress GetBasePartValue(TLinkAddress link) =>
            ↪ GetLinkReference(link).Source;
202
203         /// <summary>
204         /// <para>
205         /// Determines whether this instance first is to the left of second.
206         /// </para>
207         /// <para></para>
208         /// </summary>
209         /// <param name="firstSource">
210         /// <para>The first source.</para>
```

```
        /// <para></para>
        /// </param>
        /// <param name="firstTarget">
        /// <para>The first target.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondSource">
        /// <para>The second source.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondTarget">
        /// <para>The second target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLinkAddress firstSource, TLinkAddress
        ↪  firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget) =>
        ↪  LessThan(firstSource, secondSource) || (AreEqual(firstSource, secondSource) &&
        ↪  LessThan(firstTarget, secondTarget));

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the right of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="firstSource">
        /// <para>The first source.</para>
        /// <para></para>
        /// </param>
        /// <param name="firstTarget">
        /// <para>The first target.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondSource">
        /// <para>The second source.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondTarget">
        /// <para>The second target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLinkAddress firstSource, TLinkAddress
        ↪  firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget) =>
        ↪  GreaterThan(firstSource, secondSource) || (AreEqual(firstSource, secondSource) &&
        ↪  GreaterThan(firstTarget, secondTarget));

        /// <summary>
        /// <para>
        /// Clears the node using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLinkAddress node)
        {
            ref var link = ref GetLinkReference(node);
            link.LeftAsSource = Zero;
            link.RightAsSource = Zero;
            link.SizeAsSource = Zero;
        }
    }
}
```

```
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Memory.United.Generic
{
    /// <summary>
    /// <para>
    /// Represents the links targets avl balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksAvlBalancedTreeMethodsBase{TLinkAddress}"/>
    public unsafe class LinksTargetsAvlBalancedTreeMethods<TLinkAddress> :
    ↪ LinksAvlBalancedTreeMethodsBase<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksTargetsAvlBalancedTreeMethods"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksTargetsAvlBalancedTreeMethods(LinksConstants<TLinkAddress> constants, byte*
        ↪ links, byte* header) : base(constants, links, header) { }

        /// <summary>
        /// <para>
        /// Gets the left reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
        ↪ GetLinkReference(node).LeftAsTarget;

        /// <summary>
        /// <para>
        /// Gets the right reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
        ↪ GetLinkReference(node).RightAsTarget;

        /// <summary>
        /// <para>
        /// Gets the left using the specified node.
```

```
74          /// </para>
75          /// <para></para>
76          /// </summary>
77          /// <param name="node">
78          /// <para>The node.</para>
79          /// <para></para>
80          /// </param>
81          /// <returns>
82          /// <para>The link</para>
83          /// <para></para>
84          /// </returns>
85          [MethodImpl(MethodImplOptions.AggressiveInlining)]
86          protected override TLinkAddress GetLeft(TLinkAddress node) =>
            ↪ GetLinkReference(node).LeftAsTarget;
87
88          /// <summary>
89          /// <para>
90          /// Gets the right using the specified node.
91          /// </para>
92          /// <para></para>
93          /// </summary>
94          /// <param name="node">
95          /// <para>The node.</para>
96          /// <para></para>
97          /// </param>
98          /// <returns>
99          /// <para>The link</para>
100         /// <para></para>
101         /// </returns>
102         [MethodImpl(MethodImplOptions.AggressiveInlining)]
103         protected override TLinkAddress GetRight(TLinkAddress node) =>
            ↪ GetLinkReference(node).RightAsTarget;
104
105         /// <summary>
106         /// <para>
107         /// Sets the left using the specified node.
108         /// </para>
109         /// <para></para>
110         /// </summary>
111         /// <param name="node">
112         /// <para>The node.</para>
113         /// <para></para>
114         /// </param>
115         /// <param name="left">
116         /// <para>The left.</para>
117         /// <para></para>
118         /// </param>
119         [MethodImpl(MethodImplOptions.AggressiveInlining)]
120         protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
            ↪ GetLinkReference(node).LeftAsTarget = left;
121
122         /// <summary>
123         /// <para>
124         /// Sets the right using the specified node.
125         /// </para>
126         /// <para></para>
127         /// </summary>
128         /// <param name="node">
129         /// <para>The node.</para>
130         /// <para></para>
131         /// </param>
132         /// <param name="right">
133         /// <para>The right.</para>
134         /// <para></para>
135         /// </param>
136         [MethodImpl(MethodImplOptions.AggressiveInlining)]
137         protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
            ↪ GetLinkReference(node).RightAsTarget = right;
138
139         /// <summary>
140         /// <para>
141         /// Gets the size using the specified node.
142         /// </para>
143         /// <para></para>
144         /// </summary>
145         /// <param name="node">
146         /// <para>The node.</para>
147         /// <para></para>
```

```
148         /// </param>
149         /// <returns>
150         /// <para>The link</para>
151         /// <para></para>
152         /// </returns>
153         [MethodImpl(MethodImplOptions.AggressiveInlining)]
154         protected override TLinkAddress GetSize(TLinkAddress node) =>
            ↪ GetSizeValue(GetLinkReference(node).SizeAsTarget);
155
156         /// <summary>
157         /// <para>
158         /// Sets the size using the specified node.
159         /// </para>
160         /// <para></para>
161         /// </summary>
162         /// <param name="node">
163         /// <para>The node.</para>
164         /// <para></para>
165         /// </param>
166         /// <param name="size">
167         /// <para>The size.</para>
168         /// <para></para>
169         /// </param>
170         [MethodImpl(MethodImplOptions.AggressiveInlining)]
171         protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
            ↪ SetSizeValue(ref GetLinkReference(node).SizeAsTarget, size);
172
173         /// <summary>
174         /// <para>
175         /// Determines whether this instance get left is child.
176         /// </para>
177         /// <para></para>
178         /// </summary>
179         /// <param name="node">
180         /// <para>The node.</para>
181         /// <para></para>
182         /// </param>
183         /// <returns>
184         /// <para>The bool</para>
185         /// <para></para>
186         /// </returns>
187         [MethodImpl(MethodImplOptions.AggressiveInlining)]
188         protected override bool GetLeftIsChild(TLinkAddress node) =>
            ↪ GetLeftIsChildValue(GetLinkReference(node).SizeAsTarget);
189
190         /// <summary>
191         /// <para>
192         /// Sets the left is child using the specified node.
193         /// </para>
194         /// <para></para>
195         /// </summary>
196         /// <param name="node">
197         /// <para>The node.</para>
198         /// <para></para>
199         /// </param>
200         /// <param name="value">
201         /// <para>The value.</para>
202         /// <para></para>
203         /// </param>
204         [MethodImpl(MethodImplOptions.AggressiveInlining)]
205         protected override void SetLeftIsChild(TLinkAddress node, bool value) =>
            ↪ SetLeftIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);
206
207         /// <summary>
208         /// <para>
209         /// Determines whether this instance get right is child.
210         /// </para>
211         /// <para></para>
212         /// </summary>
213         /// <param name="node">
214         /// <para>The node.</para>
215         /// <para></para>
216         /// </param>
217         /// <returns>
218         /// <para>The bool</para>
219         /// <para></para>
220         /// </returns>
221         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
222             protected override bool GetRightIsChild(TLinkAddress node) =>
                ↪  GetRightIsChildValue(GetLinkReference(node).SizeAsTarget);
223
224             /// <summary>
225             /// <para>
226             /// Sets the right is child using the specified node.
227             /// </para>
228             /// <para></para>
229             /// </summary>
230             /// <param name="node">
231             /// <para>The node.</para>
232             /// <para></para>
233             /// </param>
234             /// <param name="value">
235             /// <para>The value.</para>
236             /// <para></para>
237             /// </param>
238             [MethodImpl(MethodImplOptions.AggressiveInlining)]
239             protected override void SetRightIsChild(TLinkAddress node, bool value) =>
                ↪  SetRightIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);
240
241             /// <summary>
242             /// <para>
243             /// Gets the balance using the specified node.
244             /// </para>
245             /// <para></para>
246             /// </summary>
247             /// <param name="node">
248             /// <para>The node.</para>
249             /// <para></para>
250             /// </param>
251             /// <returns>
252             /// <para>The sbyte</para>
253             /// <para></para>
254             /// </returns>
255             [MethodImpl(MethodImplOptions.AggressiveInlining)]
256             protected override sbyte GetBalance(TLinkAddress node) =>
                ↪  GetBalanceValue(GetLinkReference(node).SizeAsTarget);
257
258             /// <summary>
259             /// <para>
260             /// Sets the balance using the specified node.
261             /// </para>
262             /// <para></para>
263             /// </summary>
264             /// <param name="node">
265             /// <para>The node.</para>
266             /// <para></para>
267             /// </param>
268             /// <param name="value">
269             /// <para>The value.</para>
270             /// <para></para>
271             /// </param>
272             [MethodImpl(MethodImplOptions.AggressiveInlining)]
273             protected override void SetBalance(TLinkAddress node, sbyte value) =>
                ↪  SetBalanceValue(ref GetLinkReference(node).SizeAsTarget, value);
274
275             /// <summary>
276             /// <para>
277             /// Gets the tree root.
278             /// </para>
279             /// <para></para>
280             /// </summary>
281             /// <returns>
282             /// <para>The link</para>
283             /// <para></para>
284             /// </returns>
285             [MethodImpl(MethodImplOptions.AggressiveInlining)]
286             protected override TLinkAddress GetTreeRoot() => GetHeaderReference().RootAsTarget;
287
288             /// <summary>
289             /// <para>
290             /// Gets the base part value using the specified link.
291             /// </para>
292             /// <para></para>
293             /// </summary>
294             /// <param name="link">
295             /// <para>The link.</para>
```

```
296         /// <para></para>
297         /// </param>
298         /// <returns>
299         /// <para>The link</para>
300         /// <para></para>
301         /// </returns>
302         [MethodImpl(MethodImplOptions.AggressiveInlining)]
303         protected override TLinkAddress GetBasePartValue(TLinkAddress link) =>
            ↪ GetLinkReference(link).Target;
304
305         /// <summary>
306         /// <para>
307         /// Determines whether this instance first is to the left of second.
308         /// </para>
309         /// <para></para>
310         /// </summary>
311         /// <param name="firstSource">
312         /// <para>The first source.</para>
313         /// <para></para>
314         /// </param>
315         /// <param name="firstTarget">
316         /// <para>The first target.</para>
317         /// <para></para>
318         /// </param>
319         /// <param name="secondSource">
320         /// <para>The second source.</para>
321         /// <para></para>
322         /// </param>
323         /// <param name="secondTarget">
324         /// <para>The second target.</para>
325         /// <para></para>
326         /// </param>
327         /// <returns>
328         /// <para>The bool</para>
329         /// <para></para>
330         /// </returns>
331         [MethodImpl(MethodImplOptions.AggressiveInlining)]
332         protected override bool FirstIsToTheLeftOfSecond(TLinkAddress firstSource, TLinkAddress
            ↪ firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget) =>
            ↪ LessThan(firstTarget, secondTarget) || (AreEqual(firstTarget, secondTarget) &&
            ↪ LessThan(firstSource, secondSource));
333
334         /// <summary>
335         /// <para>
336         /// Determines whether this instance first is to the right of second.
337         /// </para>
338         /// <para></para>
339         /// </summary>
340         /// <param name="firstSource">
341         /// <para>The first source.</para>
342         /// <para></para>
343         /// </param>
344         /// <param name="firstTarget">
345         /// <para>The first target.</para>
346         /// <para></para>
347         /// </param>
348         /// <param name="secondSource">
349         /// <para>The second source.</para>
350         /// <para></para>
351         /// </param>
352         /// <param name="secondTarget">
353         /// <para>The second target.</para>
354         /// <para></para>
355         /// </param>
356         /// <returns>
357         /// <para>The bool</para>
358         /// <para></para>
359         /// </returns>
360         [MethodImpl(MethodImplOptions.AggressiveInlining)]
361         protected override bool FirstIsToTheRightOfSecond(TLinkAddress firstSource, TLinkAddress
            ↪ firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget) =>
            ↪ GreaterThan(firstTarget, secondTarget) || (AreEqual(firstTarget, secondTarget) &&
            ↪ GreaterThan(firstSource, secondSource));
362
363         /// <summary>
364         /// <para>
365         /// Clears the node using the specified node.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLinkAddress node)
        {
            ref var link = ref GetLinkReference(node);
            link.LeftAsTarget = Zero;
            link.RightAsTarget = Zero;
            link.SizeAsTarget = Zero;
        }
    }
}
```

1.87 ./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
{
    /// <summary>
    /// <para>
    /// Represents the links targets recursionless size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksRecursionlessSizeBalancedTreeMethodsBase{TLinkAddress}"/>
    public unsafe class LinksTargetsRecursionlessSizeBalancedTreeMethods<TLinkAddress> :
    ↪ LinksRecursionlessSizeBalancedTreeMethodsBase<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksTargetsRecursionlessSizeBalancedTreeMethods"/>
        ↪ instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<TLinkAddress>
        ↪ constants, byte* links, byte* header) : base(constants, links, header) { }

        /// <summary>
        /// <para>
        /// Gets the left reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
        ↪ GetLinkReference(node).LeftAsTarget;

        /// <summary>
        /// <para>
```

```
56          /// Gets the right reference using the specified node.
57          /// </para>
58          /// <para></para>
59          /// </summary>
60          /// <param name="node">
61          /// <para>The node.</para>
62          /// <para></para>
63          /// </param>
64          /// <returns>
65          /// <para>The ref link</para>
66          /// <para></para>
67          /// </returns>
68          [MethodImpl(MethodImplOptions.AggressiveInlining)]
69          protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
            ↪ GetLinkReference(node).RightAsTarget;
70
71          /// <summary>
72          /// <para>
73          /// Gets the left using the specified node.
74          /// </para>
75          /// <para></para>
76          /// </summary>
77          /// <param name="node">
78          /// <para>The node.</para>
79          /// <para></para>
80          /// </param>
81          /// <returns>
82          /// <para>The link</para>
83          /// <para></para>
84          /// </returns>
85          [MethodImpl(MethodImplOptions.AggressiveInlining)]
86          protected override TLinkAddress GetLeft(TLinkAddress node) =>
            ↪ GetLinkReference(node).LeftAsTarget;
87
88          /// <summary>
89          /// <para>
90          /// Gets the right using the specified node.
91          /// </para>
92          /// <para></para>
93          /// </summary>
94          /// <param name="node">
95          /// <para>The node.</para>
96          /// <para></para>
97          /// </param>
98          /// <returns>
99          /// <para>The link</para>
100         /// <para></para>
101         /// </returns>
102         [MethodImpl(MethodImplOptions.AggressiveInlining)]
103         protected override TLinkAddress GetRight(TLinkAddress node) =>
            ↪ GetLinkReference(node).RightAsTarget;
104
105         /// <summary>
106         /// <para>
107         /// Sets the left using the specified node.
108         /// </para>
109         /// <para></para>
110         /// </summary>
111         /// <param name="node">
112         /// <para>The node.</para>
113         /// <para></para>
114         /// </param>
115         /// <param name="left">
116         /// <para>The left.</para>
117         /// <para></para>
118         /// </param>
119         [MethodImpl(MethodImplOptions.AggressiveInlining)]
120         protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
            ↪ GetLinkReference(node).LeftAsTarget = left;
121
122         /// <summary>
123         /// <para>
124         /// Sets the right using the specified node.
125         /// </para>
126         /// <para></para>
127         /// </summary>
128         /// <param name="node">
129         /// <para>The node.</para>
```

```
        /// <para></para>
        /// </param>
        /// <param name="right">
        /// <para>The right.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
        ↪ GetLinkReference(node).RightAsTarget = right;

        /// <summary>
        /// <para>
        /// Gets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetSize(TLinkAddress node) =>
        ↪ GetLinkReference(node).SizeAsTarget;

        /// <summary>
        /// <para>
        /// Sets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="size">
        /// <para>The size.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
        ↪ GetLinkReference(node).SizeAsTarget = size;

        /// <summary>
        /// <para>
        /// Gets the tree root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetTreeRoot() => GetHeaderReference().RootAsTarget;

        /// <summary>
        /// <para>
        /// Gets the base part value using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetBasePartValue(TLinkAddress link) =>
        ↪ GetLinkReference(link).Target;

        /// <summary>
```

```
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="firstSource">
        /// <para>The first source.</para>
        /// <para></para>
        /// </param>
        /// <param name="firstTarget">
        /// <para>The first target.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondSource">
        /// <para>The second source.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondTarget">
        /// <para>The second target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLinkAddress firstSource, TLinkAddress
        ↪  firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget) =>
        ↪  LessThan(firstTarget, secondTarget) || (AreEqual(firstTarget, secondTarget) &&
        ↪  LessThan(firstSource, secondSource));

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the right of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="firstSource">
        /// <para>The first source.</para>
        /// <para></para>
        /// </param>
        /// <param name="firstTarget">
        /// <para>The first target.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondSource">
        /// <para>The second source.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondTarget">
        /// <para>The second target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLinkAddress firstSource, TLinkAddress
        ↪  firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget) =>
        ↪  GreaterThan(firstTarget, secondTarget) || (AreEqual(firstTarget, secondTarget) &&
        ↪  GreaterThan(firstSource, secondSource));

        /// <summary>
        /// <para>
        /// Clears the node using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLinkAddress node)
        {
            ref var link = ref GetLinkReference(node);
```

```
            link.LeftAsTarget = Zero;
            link.RightAsTarget = Zero;
            link.SizeAsTarget = Zero;
        }
    }
}
```

1.88 ./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
{
    /// <summary>
    /// <para>
    /// Represents the links targets size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksSizeBalancedTreeMethodsBase{TLinkAddress}"/>
    public unsafe class LinksTargetsSizeBalancedTreeMethods<TLinkAddress> :
    ↪ LinksSizeBalancedTreeMethodsBase<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksTargetsSizeBalancedTreeMethods"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksTargetsSizeBalancedTreeMethods(LinksConstants<TLinkAddress> constants, byte*
        ↪ links, byte* header) : base(constants, links, header) { }

        /// <summary>
        /// <para>
        /// Gets the left reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
        ↪ GetLinkReference(node).LeftAsTarget;

        /// <summary>
        /// <para>
        /// Gets the right reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref link</para>
        /// <para></para>
        /// </returns>
```

```
68          [MethodImpl(MethodImplOptions.AggressiveInlining)]
69          protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
            ↪ GetLinkReference(node).RightAsTarget;
70
71          /// <summary>
72          /// <para>
73          /// Gets the left using the specified node.
74          /// </para>
75          /// <para></para>
76          /// </summary>
77          /// <param name="node">
78          /// <para>The node.</para>
79          /// <para></para>
80          /// </param>
81          /// <returns>
82          /// <para>The link</para>
83          /// <para></para>
84          /// </returns>
85          [MethodImpl(MethodImplOptions.AggressiveInlining)]
86          protected override TLinkAddress GetLeft(TLinkAddress node) =>
            ↪ GetLinkReference(node).LeftAsTarget;
87
88          /// <summary>
89          /// <para>
90          /// Gets the right using the specified node.
91          /// </para>
92          /// <para></para>
93          /// </summary>
94          /// <param name="node">
95          /// <para>The node.</para>
96          /// <para></para>
97          /// </param>
98          /// <returns>
99          /// <para>The link</para>
100         /// <para></para>
101         /// </returns>
102         [MethodImpl(MethodImplOptions.AggressiveInlining)]
103         protected override TLinkAddress GetRight(TLinkAddress node) =>
            ↪ GetLinkReference(node).RightAsTarget;
104
105         /// <summary>
106         /// <para>
107         /// Sets the left using the specified node.
108         /// </para>
109         /// <para></para>
110         /// </summary>
111         /// <param name="node">
112         /// <para>The node.</para>
113         /// <para></para>
114         /// </param>
115         /// <param name="left">
116         /// <para>The left.</para>
117         /// <para></para>
118         /// </param>
119         [MethodImpl(MethodImplOptions.AggressiveInlining)]
120         protected override void SetLeft(TLinkAddress node, TLinkAddress left) =>
            ↪ GetLinkReference(node).LeftAsTarget = left;
121
122         /// <summary>
123         /// <para>
124         /// Sets the right using the specified node.
125         /// </para>
126         /// <para></para>
127         /// </summary>
128         /// <param name="node">
129         /// <para>The node.</para>
130         /// <para></para>
131         /// </param>
132         /// <param name="right">
133         /// <para>The right.</para>
134         /// <para></para>
135         /// </param>
136         [MethodImpl(MethodImplOptions.AggressiveInlining)]
137         protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
            ↪ GetLinkReference(node).RightAsTarget = right;
138
139         /// <summary>
```

```
        /// <para>
        /// Gets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetSize(TLinkAddress node) =>
        ↪ GetLinkReference(node).SizeAsTarget;

        /// <summary>
        /// <para>
        /// Sets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="size">
        /// <para>The size.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLinkAddress node, TLinkAddress size) =>
        ↪ GetLinkReference(node).SizeAsTarget = size;

        /// <summary>
        /// <para>
        /// Gets the tree root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetTreeRoot() => GetHeaderReference().RootAsTarget;

        /// <summary>
        /// <para>
        /// Gets the base part value using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetBasePartValue(TLinkAddress link) =>
        ↪ GetLinkReference(link).Target;

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="firstSource">
        /// <para>The first source.</para>
        /// <para></para>
        /// </param>
        /// <param name="firstTarget">
        /// <para>The first target.</para>
```

```
        /// <para></para>
        /// </param>
        /// <param name="secondSource">
        /// <para>The second source.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondTarget">
        /// <para>The second target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLinkAddress firstSource, TLinkAddress
        ↪  firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget) =>
        ↪  LessThan(firstTarget, secondTarget) || (AreEqual(firstTarget, secondTarget) &&
        ↪  LessThan(firstSource, secondSource));

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the right of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="firstSource">
        /// <para>The first source.</para>
        /// <para></para>
        /// </param>
        /// <param name="firstTarget">
        /// <para>The first target.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondSource">
        /// <para>The second source.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondTarget">
        /// <para>The second target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLinkAddress firstSource, TLinkAddress
        ↪  firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget) =>
        ↪  GreaterThan(firstTarget, secondTarget) || (AreEqual(firstTarget, secondTarget) &&
        ↪  GreaterThan(firstSource, secondSource));

        /// <summary>
        /// <para>
        /// Clears the node using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLinkAddress node)
        {
            ref var link = ref GetLinkReference(node);
            link.LeftAsTarget = Zero;
            link.RightAsTarget = Zero;
            link.SizeAsTarget = Zero;
        }
    }
}
```

1.89 ./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
{
    /// <summary>
    /// <para>
    /// Represents the united memory links.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="UnitedMemoryLinksBase{TLinkAddress}"/>
    public unsafe class UnitedMemoryLinks<TLinkAddress> : UnitedMemoryLinksBase<TLinkAddress>
        where TLinkAddress : struct
    {
        private readonly Func<ILinksTreeMethods<TLinkAddress>> _createSourceTreeMethods;
        private readonly Func<ILinksTreeMethods<TLinkAddress>> _createTargetTreeMethods;
        private byte* _header;
        private byte* _links;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UnitedMemoryLinks"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="address">
        /// <para>A address.</para>
        /// <para></para>
        /// </param>
        [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) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UnitedMemoryLinks"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="memory">
        /// <para>A memory.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory, DefaultLinksSizeStep) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UnitedMemoryLinks"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="memory">
        /// <para>A memory.</para>
        /// <para></para>
        /// </param>
        /// <param name="memoryReservationStep">
        /// <para>A memory reservation step.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnitedMemoryLinks(IResizableDirectMemory memory, long memoryReservationStep) : this(memory, memoryReservationStep, Default<LinksConstants<TLinkAddress>>.Instance, IndexTreeType.Default) { }
```

```
75
76          /// <summary>
77          /// <para>
78          /// Initializes a new <see cref="UnitedMemoryLinks"/> instance.
79          /// </para>
80          /// <para></para>
81          /// </summary>
82          /// <param name="memory">
83          /// <para>A memory.</para>
84          /// <para></para>
85          /// </param>
86          /// <param name="memoryReservationStep">
87          /// <para>A memory reservation step.</para>
88          /// <para></para>
89          /// </param>
90          /// <param name="constants">
91          /// <para>A constants.</para>
92          /// <para></para>
93          /// </param>
94          /// <param name="indexTreeType">
95          /// <para>A index tree type.</para>
96          /// <para></para>
97          /// </param>
98          [MethodImpl(MethodImplOptions.AggressiveInlining)]
99          public UnitedMemoryLinks(IResizableDirectMemory memory, long memoryReservationStep,
            ↪  LinksConstants<TLinkAddress> constants, IndexTreeType indexTreeType) : base(memory,
            ↪  memoryReservationStep, constants)
100         {
101             if (indexTreeType == IndexTreeType.SizedAndThreadedAVLBalancedTree)
102             {
103                 _createSourceTreeMethods = () => new
                    ↪  LinksSourcesAvlBalancedTreeMethods<TLinkAddress>(Constants, _links, _header);
104                 _createTargetTreeMethods = () => new
                    ↪  LinksTargetsAvlBalancedTreeMethods<TLinkAddress>(Constants, _links, _header);
105             }
106             else
107             {
108                 _createSourceTreeMethods = () => new
                    ↪  LinksSourcesSizeBalancedTreeMethods<TLinkAddress>(Constants, _links,
                    ↪  _header);
109                 _createTargetTreeMethods = () => new
                    ↪  LinksTargetsSizeBalancedTreeMethods<TLinkAddress>(Constants, _links,
                    ↪  _header);
110             }
111             Init(memory, memoryReservationStep);
112         }
113
114         /// <summary>
115         /// <para>
116         /// Sets the pointers using the specified memory.
117         /// </para>
118         /// <para></para>
119         /// </summary>
120         /// <param name="memory">
121         /// <para>The memory.</para>
122         /// <para></para>
123         /// </param>
124         [MethodImpl(MethodImplOptions.AggressiveInlining)]
125         protected override void SetPointers(IResizableDirectMemory memory)
126         {
127             _links = (byte*)memory.Pointer;
128             _header = _links;
129             SourcesTreeMethods = _createSourceTreeMethods();
130             TargetsTreeMethods = _createTargetTreeMethods();
131             UnusedLinksListMethods = new UnusedLinksListMethods<TLinkAddress>(_links, _header);
132         }
133
134         /// <summary>
135         /// <para>
136         /// Resets the pointers.
137         /// </para>
138         /// <para></para>
139         /// </summary>
140         [MethodImpl(MethodImplOptions.AggressiveInlining)]
141         protected override void ResetPointers()
142         {
143             base.ResetPointers();
```

```
            _links = null;
            _header = null;
        }

        /// <summary>
        /// <para>
        /// Gets the header reference.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A ref links header of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref LinksHeader<TLinkAddress> GetHeaderReference() => ref
        ↪  AsRef<LinksHeader<TLinkAddress>>(_header);

        /// <summary>
        /// <para>
        /// Gets the link reference using the specified link index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="linkIndex">
        /// <para>The link index.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLink<TLinkAddress> GetLinkReference(TLinkAddress linkIndex) =>
        ↪  ref AsRef<RawLink<TLinkAddress>>(_links + (LinkSizeInBytes *
        ↪  ConvertToInt64(linkIndex)));
    }
}
```

1.90 ./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;
using Platform.Delegates;

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

namespace Platform.Data.Doublets.Memory.United.Generic
{
    /// <summary>
    /// <para>
    /// Represents the united memory links base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="DisposableBase"/>
    /// <seealso cref="ILinks{TLinkAddress}"/>
    public abstract class UnitedMemoryLinksBase<TLinkAddress> : DisposableBase,
    ↪  ILinks<TLinkAddress> where TLinkAddress : struct
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;
        private static readonly Comparer<TLinkAddress> _comparer =
        ↪  Comparer<TLinkAddress>.Default;
        private static readonly UncheckedConverter<TLinkAddress, long> _addressToInt64Converter
        ↪  = UncheckedConverter<TLinkAddress, long>.Default;
        private static readonly UncheckedConverter<long, TLinkAddress> _int64ToAddressConverter
        ↪  = UncheckedConverter<long, TLinkAddress>.Default;
        private static readonly TLinkAddress _zero = default;
        private static readonly TLinkAddress _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<TLinkAddress>.SizeInBytes;

        /// <summary>
        /// <para>
        /// The size in bytes.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly long LinkHeaderSizeInBytes =
        ↪  LinksHeader<TLinkAddress>.SizeInBytes;

        /// <summary>
        /// <para>
        /// The link size in bytes.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;

        /// <summary>
        /// <para>
        /// The memory.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly IResizableDirectMemory _memory;
        /// <summary>
        /// <para>
        /// The memory reservation step.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly long _memoryReservationStep;

        /// <summary>
        /// <para>
        /// The targets tree methods.
        /// </para>
        /// <para></para>
        /// </summary>
        protected ILinksTreeMethods<TLinkAddress> TargetsTreeMethods;
        /// <summary>
        /// <para>
        /// The sources tree methods.
        /// </para>
        /// <para></para>
        /// </summary>
        protected ILinksTreeMethods<TLinkAddress> 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
        /// <summary>
        /// <para>
        /// The unused links list methods.
        /// </para>
        /// <para></para>
        /// </summary>
        protected ILinksListMethods<TLinkAddress> UnusedLinksListMethods;

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

        /// <summary>
        /// <para>
        /// Gets the constants value.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        public virtual LinksConstants<TLinkAddress> Constants
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UnitedMemoryLinksBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="memory">
        /// <para>A memory.</para>
        /// <para></para>
        /// </param>
        /// <param name="memoryReservationStep">
        /// <para>A memory reservation step.</para>
        /// <para></para>
        /// </param>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected UnitedMemoryLinksBase(IResizableDirectMemory memory, long
        ↪  memoryReservationStep, LinksConstants<TLinkAddress> constants)
        {
            _memory = memory;
            _memoryReservationStep = memoryReservationStep;
            Constants = constants;
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UnitedMemoryLinksBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="memory">
        /// <para>A memory.</para>
        /// <para></para>
        /// </param>
        /// <param name="memoryReservationStep">
        /// <para>A memory reservation step.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected UnitedMemoryLinksBase(IResizableDirectMemory memory, long
        ↪  memoryReservationStep) : this(memory, memoryReservationStep,
        ↪  Default<LinksConstants<TLinkAddress>>.Instance) { }

        /// <summary>
        /// <para>
        /// Inits the memory.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="memory">
        /// <para>The memory.</para>
        /// <para></para>
        /// </param>
        /// <param name="memoryReservationStep">
        /// <para>The memory reservation step.</para>
        /// <para></para>
        /// </param>
        [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
```

```
186             memory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) * LinkSizeInBytes) +
                ↪ LinkHeaderSizeInBytes;
187             // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
188             header.ReservedLinks = ConvertToAddress((memory.ReservedCapacity -
                ↪ LinkHeaderSizeInBytes) / LinkSizeInBytes);
189         }
190
191         /// <summary>
192         /// <para>
193         /// Counts the substitution.
194         /// </para>
195         /// <para></para>
196         /// </summary>
197         /// <param name="restriction">
198         /// <para>The substitution.</para>
199         /// <para></para>
200         /// </param>
201         /// <exception cref="NotSupportedException">
202         /// <para>Другие размеры и способы ограничений не поддерживаются.</para>
203         /// <para></para>
204         /// </exception>
205         /// <returns>
206         /// <para>The link</para>
207         /// <para></para>
208         /// </returns>
209         [MethodImpl(MethodImplOptions.AggressiveInlining)]
210         public virtual TLinkAddress Count(IList<TLinkAddress>? restriction)
211         {
212             // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
213             if (restriction.Count == 0)
214             {
215                 return Total;
216             }
217             var constants = Constants;
218             var any = constants.Any;
219             var index = restriction[constants.IndexPart];
220             if (restriction.Count == 1)
221             {
222                 if (AreEqual(index, any))
223                 {
224                     return Total;
225                 }
226                 return Exists(index) ? GetOne() : GetZero();
227             }
228             if (restriction.Count == 2)
229             {
230                 var value = restriction[1];
231                 if (AreEqual(index, any))
232                 {
233                     if (AreEqual(value, any))
234                     {
235                         return Total; // Any - как отсутствие ограничения
236                     }
237                     return Add(SourcesTreeMethods.CountUsages(value),
                        ↪ TargetsTreeMethods.CountUsages(value));
238                 }
239                 else
240                 {
241                     if (!Exists(index))
242                     {
243                         return GetZero();
244                     }
245                     if (AreEqual(value, any))
246                     {
247                         return GetOne();
248                     }
249                     ref var storedLinkValue = ref GetLinkReference(index);
250                     if (AreEqual(storedLinkValue.Source, value) ||
                        ↪ AreEqual(storedLinkValue.Target, value))
251                     {
252                         return GetOne();
253                     }
254                     return GetZero();
255                 }
256             }
257             if (restriction.Count == 3)
258             {
259                 var source = restriction[constants.SourcePart];
```

```
260                 var target = restriction[constants.TargetPart];
261                 if (AreEqual(index, any))
262                 {
263                     if (AreEqual(source, any) && AreEqual(target, any))
264                     {
265                         return Total;
266                     }
267                     else if (AreEqual(source, any))
268                     {
269                         return TargetsTreeMethods.CountUsages(target);
270                     }
271                     else if (AreEqual(target, any))
272                     {
273                         return SourcesTreeMethods.CountUsages(source);
274                     }
275                     else //if(source != Any && target != Any)
276                     {
277                         // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
278                         var link = SourcesTreeMethods.Search(source, target);
279                         return AreEqual(link, constants.Null) ? GetZero() : GetOne();
280                     }
281                 }
282                 else
283                 {
284                     if (!Exists(index))
285                     {
286                         return GetZero();
287                     }
288                     if (AreEqual(source, any) && AreEqual(target, any))
289                     {
290                         return GetOne();
291                     }
292                     ref var storedLinkValue = ref GetLinkReference(index);
293                     if (!AreEqual(source, any) && !AreEqual(target, any))
294                     {
295                         if (AreEqual(storedLinkValue.Source, source) &&
                            ↪  AreEqual(storedLinkValue.Target, target))
296                         {
297                             return GetOne();
298                         }
299                         return GetZero();
300                     }
301                     var value = default(TLinkAddress);
302                     if (AreEqual(source, any))
303                     {
304                         value = target;
305                     }
306                     if (AreEqual(target, any))
307                     {
308                         value = source;
309                     }
310                     if (AreEqual(storedLinkValue.Source, value) ||
                        ↪  AreEqual(storedLinkValue.Target, value))
311                     {
312                         return GetOne();
313                     }
314                     return GetZero();
315                 }
316             }
317             throw new NotSupportedException("Другие размеры и способы ограничений не
                ↪  поддерживаются.");
318         }
319 
320         /// <summary>
321         /// <para>
322         /// Eaches the handler.
323         /// </para>
324         /// <para></para>
325         /// </summary>
326         /// <param name="handler">
327         /// <para>The handler.</para>
328         /// <para></para>
329         /// </param>
330         /// <param name="restriction">
331         /// <para>The substitution.</para>
332         /// <para></para>
333         /// </param>
334         /// <exception cref="NotSupportedException">
```

```
335         /// <para>Другие размеры и способы ограничений не поддерживаются.</para>
336         /// <para></para>
337         /// </exception>
338         /// <returns>
339         /// <para>The link</para>
340         /// <para></para>
341         /// </returns>
342         [MethodImpl(MethodImplOptions.AggressiveInlining)]
343         public virtual TLinkAddress Each(IList<TLinkAddress>? restriction,
            ↪ ReadHandler<TLinkAddress>? handler)
344         {
345             var constants = Constants;
346             var @break = constants.Break;
347             if (restriction.Count == 0)
348             {
349                 for (var link = GetOne(); LessOrEqualThan(link,
                    ↪ GetHeaderReference().AllocatedLinks); link = Increment(link))
350                 {
351                     if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
352                     {
353                         return @break;
354                     }
355                 }
356                 return @break;
357             }
358             var @continue = constants.Continue;
359             var any = constants.Any;
360             var index = restriction[constants.IndexPart];
361             if (restriction.Count == 1)
362             {
363                 if (AreEqual(index, any))
364                 {
365                     return Each(Array.Empty<TLinkAddress>(), handler);
366                 }
367                 if (!Exists(index))
368                 {
369                     return @continue;
370                 }
371                 return handler(GetLinkStruct(index));
372             }
373             if (restriction.Count == 2)
374             {
375                 var value = restriction[1];
376                 if (AreEqual(index, any))
377                 {
378                     if (AreEqual(value, any))
379                     {
380                         return Each(Array.Empty<TLinkAddress>(), handler);
381                     }
382                     if (AreEqual(Each(new Link<TLinkAddress>(index, value, any), handler),
                        ↪ @break))
383                     {
384                         return @break;
385                     }
386                     return Each(new Link<TLinkAddress>(index, any, value), handler);
387                 }
388                 else
389                 {
390                     if (!Exists(index))
391                     {
392                         return @continue;
393                     }
394                     if (AreEqual(value, any))
395                     {
396                         return handler(GetLinkStruct(index));
397                     }
398                     ref var storedLinkValue = ref GetLinkReference(index);
399                     if (AreEqual(storedLinkValue.Source, value) ||
400                         AreEqual(storedLinkValue.Target, value))
401                     {
402                         return handler(GetLinkStruct(index));
403                     }
404                     return @continue;
405                 }
406             }
407             if (restriction.Count == 3)
408             {
409                 var source = restriction[constants.SourcePart];
```

```
410                 var target = restriction[constants.TargetPart];
411                 if (AreEqual(index, any))
412                 {
413                     if (AreEqual(source, any) && AreEqual(target, any))
414                     {
415                         return Each(Array.Empty<TLinkAddress>(), handler);
416                     }
417                     else if (AreEqual(source, any))
418                     {
419                         return TargetsTreeMethods.EachUsage(target, handler);
420                     }
421                     else if (AreEqual(target, any))
422                     {
423                         return SourcesTreeMethods.EachUsage(source, handler);
424                     }
425                     else //if(source != Any && target != Any)
426                     {
427                         var link = SourcesTreeMethods.Search(source, target);
428                         return AreEqual(link, constants.Null) ? @continue :
                            ↪ handler(GetLinkStruct(link));
429                     }
430                 }
431                 else
432                 {
433                     if (!Exists(index))
434                     {
435                         return @continue;
436                     }
437                     if (AreEqual(source, any) && AreEqual(target, any))
438                     {
439                         return handler(GetLinkStruct(index));
440                     }
441                     ref var storedLinkValue = ref GetLinkReference(index);
442                     if (!AreEqual(source, any) && !AreEqual(target, any))
443                     {
444                         if (AreEqual(storedLinkValue.Source, source) &&
445                             AreEqual(storedLinkValue.Target, target))
446                         {
447                             return handler(GetLinkStruct(index));
448                         }
449                         return @continue;
450                     }
451                     var value = default(TLinkAddress);
452                     if (AreEqual(source, any))
453                     {
454                         value = target;
455                     }
456                     if (AreEqual(target, any))
457                     {
458                         value = source;
459                     }
460                     if (AreEqual(storedLinkValue.Source, value) ||
461                         AreEqual(storedLinkValue.Target, value))
462                     {
463                         return handler(GetLinkStruct(index));
464                     }
465                     return @continue;
466                 }
467             }
468             throw new NotSupportedException("Другие размеры и способы ограничений не
                ↪ поддерживаются.");
469         }
470 
471         /// <remarks>
472         /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
                ↪ в другом месте (но не в менеджере памяти, а в логике Links)
473         /// </remarks>
474         [MethodImpl(MethodImplOptions.AggressiveInlining)]
475         public virtual TLinkAddress Update(IList<TLinkAddress>? restriction,
            ↪ IList<TLinkAddress>? substitution, WriteHandler<TLinkAddress>? handler)
476         {
477             var constants = Constants;
478             var @null = constants.Null;
479             var linkIndex = restriction[constants.IndexPart];
480             var before = GetLinkStruct(linkIndex);
481             ref var link = ref GetLinkReference(linkIndex);
482             ref var header = ref GetHeaderReference();
483             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 handler?.Invoke(before, GetLinkStruct(linkIndex)) ?? Constants.Continue;
        }

        /// <remarks>
        /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет пространство
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLinkAddress Create(IList<TLinkAddress>? substitution, WriteHandler<TLinkAddress>? handler)
        {
            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<TLinkAddress>(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 handler?.Invoke(null, new Link<TLinkAddress>(freeLink, Constants.Null, Constants.Null)) ?? Constants.Continue;
        }

        /// <summary>
        /// <para>
        /// Deletes the substitution.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restriction">
        /// <para>The substitution.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLinkAddress Delete(IList<TLinkAddress>? restriction, WriteHandler<TLinkAddress>? handler)
        {
            ref var header = ref GetHeaderReference();
            var link = restriction[Constants.IndexPart];
            var before = GetLinkStruct(link);
```

```
555                 if (LessThan(link, header.AllocatedLinks))
556                 {
557                     UnusedLinksListMethods.AttachAsFirst(link);
558                     return handler?.Invoke(before, null) ?? Constants.Continue;
559                 }
560                 else if (AreEqual(link, header.AllocatedLinks))
561                 {
562                     header.AllocatedLinks = Decrement(header.AllocatedLinks);
563                     _memory.UsedCapacity -= LinkSizeInBytes;
564                     // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
                        ↪  пока не дойдём до первой существующей связи
565                     // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
566                     while (GreaterThan(header.AllocatedLinks, GetZero()) &&
                        ↪  IsUnusedLink(header.AllocatedLinks))
567                     {
568                         UnusedLinksListMethods.Detach(header.AllocatedLinks);
569                         header.AllocatedLinks = Decrement(header.AllocatedLinks);
570                         _memory.UsedCapacity -= LinkSizeInBytes;
571                     }
572                     return handler?.Invoke(before, null) ?? Constants.Continue;
573                 }
574                 return Constants.Continue;
575             }
576 
577             /// <summary>
578             /// <para>
579             /// Gets the link struct using the specified link index.
580             /// </para>
581             /// <para></para>
582             /// </summary>
583             /// <param name="linkIndex">
584             /// <para>The link index.</para>
585             /// <para></para>
586             /// </param>
587             /// <returns>
588             /// <para>A list of t link</para>
589             /// <para></para>
590             /// </returns>
591             [MethodImpl(MethodImplOptions.AggressiveInlining)]
592             public IList<TLinkAddress>? GetLinkStruct(TLinkAddress linkIndex)
593             {
594                 ref var link = ref GetLinkReference(linkIndex);
595                 return new Link<TLinkAddress>(linkIndex, link.Source, link.Target);
596             }
597 
598             /// <remarks>
599             /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
                ↪  адрес реально поменялся
600             ///
601             /// Указатель this.links может быть в том же месте,
602             /// так как 0-я связь не используется и имеет такой же размер как Header,
603             /// поэтому header размещается в том же месте, что и 0-я связь
604             /// </remarks>
605             [MethodImpl(MethodImplOptions.AggressiveInlining)]
606             protected abstract void SetPointers(IResizableDirectMemory memory);
607 
608             /// <summary>
609             /// <para>
610             /// Resets the pointers.
611             /// </para>
612             /// <para></para>
613             /// </summary>
614             [MethodImpl(MethodImplOptions.AggressiveInlining)]
615             protected virtual void ResetPointers()
616             {
617                 SourcesTreeMethods = null;
618                 TargetsTreeMethods = null;
619                 UnusedLinksListMethods = null;
620             }
621 
622             /// <summary>
623             /// <para>
624             /// Gets the header reference.
625             /// </para>
626             /// <para></para>
627             /// </summary>
628             /// <returns>
629             /// <para>A ref links header of t link</para>
```

```
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract ref LinksHeader<TLinkAddress> GetHeaderReference();

        /// <summary>
        /// <para>
        /// Gets the link reference using the specified link index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="linkIndex">
        /// <para>The link index.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract ref RawLink<TLinkAddress> GetLinkReference(TLinkAddress linkIndex);

        /// <summary>
        /// <para>
        /// Determines whether this instance exists.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool Exists(TLinkAddress link)
            => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
            && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
            && !IsUnusedLink(link);

        /// <summary>
        /// <para>
        /// Determines whether this instance is unused link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="linkIndex">
        /// <para>The link index.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool IsUnusedLink(TLinkAddress 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;
            }
        }

        /// <summary>
        /// <para>
        /// Gets the one.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
```

```
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLinkAddress GetOne() => _one;

        /// <summary>
        /// <para>
        /// Gets the zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLinkAddress GetZero() => default;

        /// <summary>
        /// <para>
        /// Determines whether this instance are equal.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool AreEqual(TLinkAddress first, TLinkAddress second) =>
        ↪  _equalityComparer.Equals(first, second);

        /// <summary>
        /// <para>
        /// Determines whether this instance less than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool LessThan(TLinkAddress first, TLinkAddress second) =>
        ↪  _comparer.Compare(first, second) < 0;

        /// <summary>
        /// <para>
        /// Determines whether this instance less or equal than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
```

```
783         /// <para>The bool</para>
784         /// <para></para>
785         /// </returns>
786         [MethodImpl(MethodImplOptions.AggressiveInlining)]
787         protected virtual bool LessOrEqualThan(TLinkAddress first, TLinkAddress second) =>
            ↪  _comparer.Compare(first, second) <= 0;
788
789         /// <summary>
790         /// <para>
791         /// Determines whether this instance greater than.
792         /// </para>
793         /// <para></para>
794         /// </summary>
795         /// <param name="first">
796         /// <para>The first.</para>
797         /// <para></para>
798         /// </param>
799         /// <param name="second">
800         /// <para>The second.</para>
801         /// <para></para>
802         /// </param>
803         /// <returns>
804         /// <para>The bool</para>
805         /// <para></para>
806         /// </returns>
807         [MethodImpl(MethodImplOptions.AggressiveInlining)]
808         protected virtual bool GreaterThan(TLinkAddress first, TLinkAddress second) =>
            ↪  _comparer.Compare(first, second) > 0;
809
810         /// <summary>
811         /// <para>
812         /// Determines whether this instance greater or equal than.
813         /// </para>
814         /// <para></para>
815         /// </summary>
816         /// <param name="first">
817         /// <para>The first.</para>
818         /// <para></para>
819         /// </param>
820         /// <param name="second">
821         /// <para>The second.</para>
822         /// <para></para>
823         /// </param>
824         /// <returns>
825         /// <para>The bool</para>
826         /// <para></para>
827         /// </returns>
828         [MethodImpl(MethodImplOptions.AggressiveInlining)]
829         protected virtual bool GreaterOrEqualThan(TLinkAddress first, TLinkAddress second) =>
            ↪  _comparer.Compare(first, second) >= 0;
830
831         /// <summary>
832         /// <para>
833         /// Converts the to int 64 using the specified value.
834         /// </para>
835         /// <para></para>
836         /// </summary>
837         /// <param name="value">
838         /// <para>The value.</para>
839         /// <para></para>
840         /// </param>
841         /// <returns>
842         /// <para>The long</para>
843         /// <para></para>
844         /// </returns>
845         [MethodImpl(MethodImplOptions.AggressiveInlining)]
846         protected virtual long ConvertToInt64(TLinkAddress value) =>
            ↪  _addressToInt64Converter.Convert(value);
847
848         /// <summary>
849         /// <para>
850         /// Converts the to address using the specified value.
851         /// </para>
852         /// <para></para>
853         /// </summary>
854         /// <param name="value">
855         /// <para>The value.</para>
856         /// <para></para>
```

```
857             /// </param>
858             /// <returns>
859             /// <para>The link</para>
860             /// <para></para>
861             /// </returns>
862             [MethodImpl(MethodImplOptions.AggressiveInlining)]
863             protected virtual TLinkAddress ConvertToAddress(long value) =>
                ↪ _int64ToAddressConverter.Convert(value);
864
865             /// <summary>
866             /// <para>
867             /// Adds the first.
868             /// </para>
869             /// <para></para>
870             /// </summary>
871             /// <param name="first">
872             /// <para>The first.</para>
873             /// <para></para>
874             /// </param>
875             /// <param name="second">
876             /// <para>The second.</para>
877             /// <para></para>
878             /// </param>
879             /// <returns>
880             /// <para>The link</para>
881             /// <para></para>
882             /// </returns>
883             [MethodImpl(MethodImplOptions.AggressiveInlining)]
884             protected virtual TLinkAddress Add(TLinkAddress first, TLinkAddress second) =>
                ↪ Arithmetic<TLinkAddress>.Add(first, second);
885
886             /// <summary>
887             /// <para>
888             /// Subtracts the first.
889             /// </para>
890             /// <para></para>
891             /// </summary>
892             /// <param name="first">
893             /// <para>The first.</para>
894             /// <para></para>
895             /// </param>
896             /// <param name="second">
897             /// <para>The second.</para>
898             /// <para></para>
899             /// </param>
900             /// <returns>
901             /// <para>The link</para>
902             /// <para></para>
903             /// </returns>
904             [MethodImpl(MethodImplOptions.AggressiveInlining)]
905             protected virtual TLinkAddress Subtract(TLinkAddress first, TLinkAddress second) =>
                ↪ Arithmetic<TLinkAddress>.Subtract(first, second);
906
907             /// <summary>
908             /// <para>
909             /// Increments the link.
910             /// </para>
911             /// <para></para>
912             /// </summary>
913             /// <param name="link">
914             /// <para>The link.</para>
915             /// <para></para>
916             /// </param>
917             /// <returns>
918             /// <para>The link</para>
919             /// <para></para>
920             /// </returns>
921             [MethodImpl(MethodImplOptions.AggressiveInlining)]
922             protected virtual TLinkAddress Increment(TLinkAddress link) =>
                ↪ Arithmetic<TLinkAddress>.Increment(link);
923
924             /// <summary>
925             /// <para>
926             /// Decrements the link.
927             /// </para>
928             /// <para></para>
929             /// </summary>
930             /// <param name="link">
```

```
931         /// <para>The link.</para>
932         /// <para></para>
933         /// </param>
934         /// <returns>
935         /// <para>The link</para>
936         /// <para></para>
937         /// </returns>
938         [MethodImpl(MethodImplOptions.AggressiveInlining)]
939         protected virtual TLinkAddress Decrement(TLinkAddress link) =>
            ↪ Arithmetic<TLinkAddress>.Decrement(link);
940
941         #region Disposable
942
943         /// <summary>
944         /// <para>
945         /// Gets the allow multiple dispose calls value.
946         /// </para>
947         /// <para></para>
948         /// </summary>
949         protected override bool AllowMultipleDisposeCalls
950         {
951             [MethodImpl(MethodImplOptions.AggressiveInlining)]
952             get => true;
953         }
954
955         /// <summary>
956         /// <para>
957         /// Disposes the manual.
958         /// </para>
959         /// <para></para>
960         /// </summary>
961         /// <param name="manual">
962         /// <para>The manual.</para>
963         /// <para></para>
964         /// </param>
965         /// <param name="wasDisposed">
966         /// <para>The was disposed.</para>
967         /// <para></para>
968         /// </param>
969         [MethodImpl(MethodImplOptions.AggressiveInlining)]
970         protected override void Dispose(bool manual, bool wasDisposed)
971         {
972             if (!wasDisposed)
973             {
974                 ResetPointers();
975                 _memory.DisposeIfPossible();
976             }
977         }
978
979         #endregion
980     }
981 }
```

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

```
1  using System.Runtime.CompilerServices;
2  using Platform.Collections.Methods.Lists;
3  using Platform.Converters;
4  using static System.Runtime.CompilerServices.Unsafe;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Memory.United.Generic
9  {
10     /// <summary>
11     /// <para>
12     /// Represents the unused links list methods.
13     /// </para>
14     /// <para></para>
15     /// </summary>
16     /// <seealso cref="AbsoluteCircularDoublyLinkedListMethods{TLinkAddress}"/>
17     /// <seealso cref="ILinksListMethods{TLinkAddress}"/>
18     public unsafe class UnusedLinksListMethods<TLinkAddress> :
       ↪ AbsoluteCircularDoublyLinkedListMethods<TLinkAddress>, ILinksListMethods<TLinkAddress>
19     {
20         private static readonly UncheckedConverter<TLinkAddress, long> _addressToInt64Converter
           ↪ = UncheckedConverter<TLinkAddress, long>.Default;
21         private readonly byte* _links;
22         private readonly byte* _header;
23
```

```
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UnusedLinksListMethods"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnusedLinksListMethods(byte* links, byte* header)
        {
            _links = links;
            _header = header;
        }

        /// <summary>
        /// <para>
        /// Gets the header reference.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A ref links header of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref LinksHeader<TLinkAddress> GetHeaderReference() => ref
        ↪  AsRef<LinksHeader<TLinkAddress>>(_header);

        /// <summary>
        /// <para>
        /// Gets the link reference using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref RawLink<TLinkAddress> GetLinkReference(TLinkAddress link) => ref
        ↪  AsRef<RawLink<TLinkAddress>>(_links + (RawLink<TLinkAddress>.SizeInBytes *
        ↪  _addressToInt64Converter.Convert(link)));

        /// <summary>
        /// <para>
        /// Gets the first.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetFirst() => GetHeaderReference().FirstFreeLink;

        /// <summary>
        /// <para>
        /// Gets the last.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
99          protected override TLinkAddress GetLast() => GetHeaderReference().LastFreeLink;
100
101         /// <summary>
102         /// <para>
103         /// Gets the previous using the specified element.
104         /// </para>
105         /// <para></para>
106         /// </summary>
107         /// <param name="element">
108         /// <para>The element.</para>
109         /// <para></para>
110         /// </param>
111         /// <returns>
112         /// <para>The link</para>
113         /// <para></para>
114         /// </returns>
115         [MethodImpl(MethodImplOptions.AggressiveInlining)]
116         protected override TLinkAddress GetPrevious(TLinkAddress element) =>
            ↪ GetLinkReference(element).Source;
117
118         /// <summary>
119         /// <para>
120         /// Gets the next using the specified element.
121         /// </para>
122         /// <para></para>
123         /// </summary>
124         /// <param name="element">
125         /// <para>The element.</para>
126         /// <para></para>
127         /// </param>
128         /// <returns>
129         /// <para>The link</para>
130         /// <para></para>
131         /// </returns>
132         [MethodImpl(MethodImplOptions.AggressiveInlining)]
133         protected override TLinkAddress GetNext(TLinkAddress element) =>
            ↪ GetLinkReference(element).Target;
134
135         /// <summary>
136         /// <para>
137         /// Gets the size.
138         /// </para>
139         /// <para></para>
140         /// </summary>
141         /// <returns>
142         /// <para>The link</para>
143         /// <para></para>
144         /// </returns>
145         [MethodImpl(MethodImplOptions.AggressiveInlining)]
146         protected override TLinkAddress GetSize() => GetHeaderReference().FreeLinks;
147
148         /// <summary>
149         /// <para>
150         /// Sets the first using the specified element.
151         /// </para>
152         /// <para></para>
153         /// </summary>
154         /// <param name="element">
155         /// <para>The element.</para>
156         /// <para></para>
157         /// </param>
158         [MethodImpl(MethodImplOptions.AggressiveInlining)]
159         protected override void SetFirst(TLinkAddress element) =>
            ↪ GetHeaderReference().FirstFreeLink = element;
160
161         /// <summary>
162         /// <para>
163         /// Sets the last using the specified element.
164         /// </para>
165         /// <para></para>
166         /// </summary>
167         /// <param name="element">
168         /// <para>The element.</para>
169         /// <para></para>
170         /// </param>
171         [MethodImpl(MethodImplOptions.AggressiveInlining)]
172         protected override void SetLast(TLinkAddress element) =>
            ↪ GetHeaderReference().LastFreeLink = element;
```

```

        /// <summary>
        /// <para>
        /// Sets the previous using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <param name="previous">
        /// <para>The previous.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetPrevious(TLinkAddress element, TLinkAddress previous) =>
        ↪  GetLinkReference(element).Source = previous;

        /// <summary>
        /// <para>
        /// Sets the next using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <param name="next">
        /// <para>The next.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetNext(TLinkAddress element, TLinkAddress next) =>
        ↪  GetLinkReference(element).Target = next;

        /// <summary>
        /// <para>
        /// Sets the size using the specified size.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="size">
        /// <para>The size.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLinkAddress size) => GetHeaderReference().FreeLinks =
        ↪  size;
    }
}
```

1.92 ./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
{
    /// <summary>
    /// <para>
    /// The raw link.
    /// </para>
    /// <para></para>
    /// </summary>
    public struct RawLink<TLinkAddress> : IEquatable<RawLink<TLinkAddress>>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;

        /// <summary>
        /// <para>
        /// The size.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        public static readonly long SizeInBytes = Structure<RawLink<TLinkAddress>>.Size;

        /// <summary>
        /// <para>
        /// The source.
        /// </para>
        /// <para></para>
        /// </summary>
        public TLinkAddress Source;
        /// <summary>
        /// <para>
        /// The target.
        /// </para>
        /// <para></para>
        /// </summary>
        public TLinkAddress Target;
        /// <summary>
        /// <para>
        /// The left as source.
        /// </para>
        /// <para></para>
        /// </summary>
        public TLinkAddress LeftAsSource;
        /// <summary>
        /// <para>
        /// The right as source.
        /// </para>
        /// <para></para>
        /// </summary>
        public TLinkAddress RightAsSource;
        /// <summary>
        /// <para>
        /// The size as source.
        /// </para>
        /// <para></para>
        /// </summary>
        public TLinkAddress SizeAsSource;
        /// <summary>
        /// <para>
        /// The left as target.
        /// </para>
        /// <para></para>
        /// </summary>
        public TLinkAddress LeftAsTarget;
        /// <summary>
        /// <para>
        /// The right as target.
        /// </para>
        /// <para></para>
        /// </summary>
        public TLinkAddress RightAsTarget;
        /// <summary>
        /// <para>
        /// The size as target.
        /// </para>
        /// <para></para>
        /// </summary>
        public TLinkAddress SizeAsTarget;

        /// <summary>
        /// <para>
        /// Determines whether this instance equals.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="obj">
        /// <para>The obj.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Equals(object obj) => obj is RawLink<TLinkAddress> link ?
        ↪  Equals(link) : false;
```

```
101
102         /// <summary>
103         /// <para>
104         /// Determines whether this instance equals.
105         /// </para>
106         /// <para></para>
107         /// </summary>
108         /// <param name="other">
109         /// <para>The other.</para>
110         /// <para></para>
111         /// </param>
112         /// <returns>
113         /// <para>The bool</para>
114         /// <para></para>
115         /// </returns>
116         [MethodImpl(MethodImplOptions.AggressiveInlining)]
117         public bool Equals(RawLink<TLinkAddress> other)
118             => _equalityComparer.Equals(Source, other.Source)
119             && _equalityComparer.Equals(Target, other.Target)
120             && _equalityComparer.Equals(LeftAsSource, other.LeftAsSource)
121             && _equalityComparer.Equals(RightAsSource, other.RightAsSource)
122             && _equalityComparer.Equals(SizeAsSource, other.SizeAsSource)
123             && _equalityComparer.Equals(LeftAsTarget, other.LeftAsTarget)
124             && _equalityComparer.Equals(RightAsTarget, other.RightAsTarget)
125             && _equalityComparer.Equals(SizeAsTarget, other.SizeAsTarget);
126
127         /// <summary>
128         /// <para>
129         /// Gets the hash code.
130         /// </para>
131         /// <para></para>
132         /// </summary>
133         /// <returns>
134         /// <para>The int</para>
135         /// <para></para>
136         /// </returns>
137         [MethodImpl(MethodImplOptions.AggressiveInlining)]
138         public override int GetHashCode() => (Source, Target, LeftAsSource, RightAsSource,
            ↪  SizeAsSource, LeftAsTarget, RightAsTarget, SizeAsTarget).GetHashCode();
139
140         [MethodImpl(MethodImplOptions.AggressiveInlining)]
141         public static bool operator ==(RawLink<TLinkAddress> left, RawLink<TLinkAddress> right)
            ↪  => left.Equals(right);
142
143         [MethodImpl(MethodImplOptions.AggressiveInlining)]
144         public static bool operator !=(RawLink<TLinkAddress> left, RawLink<TLinkAddress> right)
            ↪  => !(left == right);
145     }
146 }
```

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

```
1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.Memory.United.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.United.Specific
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the int 32 links recursionless size balanced tree methods base.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     /// <seealso cref="LinksRecursionlessSizeBalancedTreeMethodsBase{uint}"/>
15     public unsafe abstract class UInt32LinksRecursionlessSizeBalancedTreeMethodsBase :
        ↪  LinksRecursionlessSizeBalancedTreeMethodsBase<uint>
16     {
17         /// <summary>
18         /// <para>
19         /// The links.
20         /// </para>
21         /// <para></para>
22         /// </summary>
23         protected new readonly RawLink<uint>* Links;
24         /// <summary>
25         /// <para>
26         /// The header.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        protected new readonly LinksHeader<uint>* Header;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt32LinksRecursionlessSizeBalancedTreeMethodsBase"/>
        ↪  instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        protected UInt32LinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<uint>
        ↪  constants, RawLink<uint>* links, LinksHeader<uint>* header)
            : base(constants, (byte*)links, (byte*)header)
        {
            Links = links;
            Header = header;
        }

        /// <summary>
        /// <para>
        /// Gets the zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetZero() => 0U;

        /// <summary>
        /// <para>
        /// Determines whether this instance equal to zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool EqualToZero(uint value) => value == 0U;

        /// <summary>
        /// <para>
        /// Determines whether this instance are equal.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
```

```
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool AreEqual(uint first, uint second) => first == second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThanZero(uint value) => value > OU;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThan(uint first, uint second) => first > second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater or equal than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterOrEqualThan(uint first, uint second) => first >= second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater or equal than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
```

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

        /// <summary>
        /// <para>
        /// Determines whether this instance less or equal than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessOrEqualThanZero(uint value) => value == 0U; // value is
        ↪ always >= 0 for uint

        /// <summary>
        /// <para>
        /// Determines whether this instance less or equal than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessOrEqualThan(uint first, uint second) => first <= second;

        /// <summary>
        /// <para>
        /// Determines whether this instance less than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessThanZero(uint value) => false; // value < 0 is always false
        ↪ for uint

        /// <summary>
        /// <para>
        /// Determines whether this instance less than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
```

```
256         /// </returns>
257         [MethodImpl(MethodImplOptions.AggressiveInlining)]
258         protected override bool LessThan(uint first, uint second) => first < second;
259
260         /// <summary>
261         /// <para>
262         /// Increments the value.
263         /// </para>
264         /// <para></para>
265         /// </summary>
266         /// <param name="value">
267         /// <para>The value.</para>
268         /// <para></para>
269         /// </param>
270         /// <returns>
271         /// <para>The uint</para>
272         /// <para></para>
273         /// </returns>
274         [MethodImpl(MethodImplOptions.AggressiveInlining)]
275         protected override uint Increment(uint value) => ++value;
276
277         /// <summary>
278         /// <para>
279         /// Decrements the value.
280         /// </para>
281         /// <para></para>
282         /// </summary>
283         /// <param name="value">
284         /// <para>The value.</para>
285         /// <para></para>
286         /// </param>
287         /// <returns>
288         /// <para>The uint</para>
289         /// <para></para>
290         /// </returns>
291         [MethodImpl(MethodImplOptions.AggressiveInlining)]
292         protected override uint Decrement(uint value) => --value;
293
294         /// <summary>
295         /// <para>
296         /// Adds the first.
297         /// </para>
298         /// <para></para>
299         /// </summary>
300         /// <param name="first">
301         /// <para>The first.</para>
302         /// <para></para>
303         /// </param>
304         /// <param name="second">
305         /// <para>The second.</para>
306         /// <para></para>
307         /// </param>
308         /// <returns>
309         /// <para>The uint</para>
310         /// <para></para>
311         /// </returns>
312         [MethodImpl(MethodImplOptions.AggressiveInlining)]
313         protected override uint Add(uint first, uint second) => first + second;
314
315         /// <summary>
316         /// <para>
317         /// Subtracts the first.
318         /// </para>
319         /// <para></para>
320         /// </summary>
321         /// <param name="first">
322         /// <para>The first.</para>
323         /// <para></para>
324         /// </param>
325         /// <param name="second">
326         /// <para>The second.</para>
327         /// <para></para>
328         /// </param>
329         /// <returns>
330         /// <para>The uint</para>
331         /// <para></para>
332         /// </returns>
333         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
334         protected override uint Subtract(uint first, uint second) => first - second;
335
336         /// <summary>
337         /// <para>
338         /// Determines whether this instance first is to the left of second.
339         /// </para>
340         /// <para></para>
341         /// </summary>
342         /// <param name="first">
343         /// <para>The first.</para>
344         /// <para></para>
345         /// </param>
346         /// <param name="second">
347         /// <para>The second.</para>
348         /// <para></para>
349         /// </param>
350         /// <returns>
351         /// <para>The bool</para>
352         /// <para></para>
353         /// </returns>
354         [MethodImpl(MethodImplOptions.AggressiveInlining)]
355         protected override bool FirstIsToTheLeftOfSecond(uint first, uint second)
356         {
357             ref var firstLink = ref Links[first];
358             ref var secondLink = ref Links[second];
359             return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
360         }
361
362         /// <summary>
363         /// <para>
364         /// Determines whether this instance first is to the right of second.
365         /// </para>
366         /// <para></para>
367         /// </summary>
368         /// <param name="first">
369         /// <para>The first.</para>
370         /// <para></para>
371         /// </param>
372         /// <param name="second">
373         /// <para>The second.</para>
374         /// <para></para>
375         /// </param>
376         /// <returns>
377         /// <para>The bool</para>
378         /// <para></para>
379         /// </returns>
380         [MethodImpl(MethodImplOptions.AggressiveInlining)]
381         protected override bool FirstIsToTheRightOfSecond(uint first, uint second)
382         {
383             ref var firstLink = ref Links[first];
384             ref var secondLink = ref Links[second];
385             return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
386         }
387
388         /// <summary>
389         /// <para>
390         /// Gets the header reference.
391         /// </para>
392         /// <para></para>
393         /// </summary>
394         /// <returns>
395         /// <para>A ref links header of uint</para>
396         /// <para></para>
397         /// </returns>
398         [MethodImpl(MethodImplOptions.AggressiveInlining)]
399         protected override ref LinksHeader<uint> GetHeaderReference() => ref *Header;
400
401         /// <summary>
402         /// <para>
403         /// Gets the link reference using the specified link.
404         /// </para>
405         /// <para></para>
406         /// </summary>
407         /// <param name="link">
408         /// <para>The link.</para>
409         /// <para></para>
```

```
410         /// </param>
411         /// <returns>
412         /// <para>A ref raw link of uint</para>
413         /// <para></para>
414         /// </returns>
415         [MethodImpl(MethodImplOptions.AggressiveInlining)]
416         protected override ref RawLink<uint> GetLinkReference(uint link) => ref Links[link];
417     }
418 }
```

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

```
1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.Memory.United.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.United.Specific
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the int 32 links size balanced tree methods base.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     /// <seealso cref="LinksSizeBalancedTreeMethodsBase{uint}"/>
15     public unsafe abstract class UInt32LinksSizeBalancedTreeMethodsBase :
       ↪  LinksSizeBalancedTreeMethodsBase<uint>
16     {
17         /// <summary>
18         /// <para>
19         /// The links.
20         /// </para>
21         /// <para></para>
22         /// </summary>
23         protected new readonly RawLink<uint>* Links;
24         /// <summary>
25         /// <para>
26         /// The header.
27         /// </para>
28         /// <para></para>
29         /// </summary>
30         protected new readonly LinksHeader<uint>* Header;
31
32         /// <summary>
33         /// <para>
34         /// Initializes a new <see cref="UInt32LinksSizeBalancedTreeMethodsBase"/> instance.
35         /// </para>
36         /// <para></para>
37         /// </summary>
38         /// <param name="constants">
39         /// <para>A constants.</para>
40         /// <para></para>
41         /// </param>
42         /// <param name="links">
43         /// <para>A links.</para>
44         /// <para></para>
45         /// </param>
46         /// <param name="header">
47         /// <para>A header.</para>
48         /// <para></para>
49         /// </param>
50         protected UInt32LinksSizeBalancedTreeMethodsBase(LinksConstants<uint> constants,
           ↪  RawLink<uint>* links, LinksHeader<uint>* header)
51             : base(constants, (byte*)links, (byte*)header)
52         {
53             Links = links;
54             Header = header;
55         }
56
57         /// <summary>
58         /// <para>
59         /// Gets the zero.
60         /// </para>
61         /// <para></para>
62         /// </summary>
63         /// <returns>
64         /// <para>The uint</para>
65         /// <para></para>
66         /// </returns>
```

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

        /// <summary>
        /// <para>
        /// Determines whether this instance equal to zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool EqualToZero(uint value) => value == 0U;

        /// <summary>
        /// <para>
        /// Determines whether this instance are equal.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool AreEqual(uint first, uint second) => first == second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThanZero(uint value) => value > 0U;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThan(uint first, uint second) => first > second;
```

```
145
146         /// <summary>
147         /// <para>
148         /// Determines whether this instance greater or equal than.
149         /// </para>
150         /// <para></para>
151         /// </summary>
152         /// <param name="first">
153         /// <para>The first.</para>
154         /// <para></para>
155         /// </param>
156         /// <param name="second">
157         /// <para>The second.</para>
158         /// <para></para>
159         /// </param>
160         /// <returns>
161         /// <para>The bool</para>
162         /// <para></para>
163         /// </returns>
164         [MethodImpl(MethodImplOptions.AggressiveInlining)]
165         protected override bool GreaterOrEqualThan(uint first, uint second) => first >= second;
166
167         /// <summary>
168         /// <para>
169         /// Determines whether this instance greater or equal than zero.
170         /// </para>
171         /// <para></para>
172         /// </summary>
173         /// <param name="value">
174         /// <para>The value.</para>
175         /// <para></para>
176         /// </param>
177         /// <returns>
178         /// <para>The bool</para>
179         /// <para></para>
180         /// </returns>
181         [MethodImpl(MethodImplOptions.AggressiveInlining)]
182         protected override bool GreaterOrEqualThanZero(uint value) => true; // value >= 0 is
            ↪  always true for uint
183
184         /// <summary>
185         /// <para>
186         /// Determines whether this instance less or equal than zero.
187         /// </para>
188         /// <para></para>
189         /// </summary>
190         /// <param name="value">
191         /// <para>The value.</para>
192         /// <para></para>
193         /// </param>
194         /// <returns>
195         /// <para>The bool</para>
196         /// <para></para>
197         /// </returns>
198         [MethodImpl(MethodImplOptions.AggressiveInlining)]
199         protected override bool LessOrEqualThanZero(uint value) => value == 0U; // value is
            ↪  always >= 0 for uint
200
201         /// <summary>
202         /// <para>
203         /// Determines whether this instance less or equal than.
204         /// </para>
205         /// <para></para>
206         /// </summary>
207         /// <param name="first">
208         /// <para>The first.</para>
209         /// <para></para>
210         /// </param>
211         /// <param name="second">
212         /// <para>The second.</para>
213         /// <para></para>
214         /// </param>
215         /// <returns>
216         /// <para>The bool</para>
217         /// <para></para>
218         /// </returns>
219         [MethodImpl(MethodImplOptions.AggressiveInlining)]
220         protected override bool LessOrEqualThan(uint first, uint second) => first <= second;
```

```
221
222         /// <summary>
223         /// <para>
224         /// Determines whether this instance less than zero.
225         /// </para>
226         /// <para></para>
227         /// </summary>
228         /// <param name="value">
229         /// <para>The value.</para>
230         /// <para></para>
231         /// </param>
232         /// <returns>
233         /// <para>The bool</para>
234         /// <para></para>
235         /// </returns>
236         [MethodImpl(MethodImplOptions.AggressiveInlining)]
237         protected override bool LessThanZero(uint value) => false; // value < 0 is always false
            ↪  for uint
238
239         /// <summary>
240         /// <para>
241         /// Determines whether this instance less than.
242         /// </para>
243         /// <para></para>
244         /// </summary>
245         /// <param name="first">
246         /// <para>The first.</para>
247         /// <para></para>
248         /// </param>
249         /// <param name="second">
250         /// <para>The second.</para>
251         /// <para></para>
252         /// </param>
253         /// <returns>
254         /// <para>The bool</para>
255         /// <para></para>
256         /// </returns>
257         [MethodImpl(MethodImplOptions.AggressiveInlining)]
258         protected override bool LessThan(uint first, uint second) => first < second;
259
260         /// <summary>
261         /// <para>
262         /// Increments the value.
263         /// </para>
264         /// <para></para>
265         /// </summary>
266         /// <param name="value">
267         /// <para>The value.</para>
268         /// <para></para>
269         /// </param>
270         /// <returns>
271         /// <para>The uint</para>
272         /// <para></para>
273         /// </returns>
274         [MethodImpl(MethodImplOptions.AggressiveInlining)]
275         protected override uint Increment(uint value) => ++value;
276
277         /// <summary>
278         /// <para>
279         /// Decrements the value.
280         /// </para>
281         /// <para></para>
282         /// </summary>
283         /// <param name="value">
284         /// <para>The value.</para>
285         /// <para></para>
286         /// </param>
287         /// <returns>
288         /// <para>The uint</para>
289         /// <para></para>
290         /// </returns>
291         [MethodImpl(MethodImplOptions.AggressiveInlining)]
292         protected override uint Decrement(uint value) => --value;
293
294         /// <summary>
295         /// <para>
296         /// Adds the first.
297         /// </para>
```

```
298             /// <para></para>
299             /// </summary>
300             /// <param name="first">
301             /// <para>The first.</para>
302             /// <para></para>
303             /// </param>
304             /// <param name="second">
305             /// <para>The second.</para>
306             /// <para></para>
307             /// </param>
308             /// <returns>
309             /// <para>The uint</para>
310             /// <para></para>
311             /// </returns>
312             [MethodImpl(MethodImplOptions.AggressiveInlining)]
313             protected override uint Add(uint first, uint second) => first + second;
314
315             /// <summary>
316             /// <para>
317             /// Subtracts the first.
318             /// </para>
319             /// <para></para>
320             /// </summary>
321             /// <param name="first">
322             /// <para>The first.</para>
323             /// <para></para>
324             /// </param>
325             /// <param name="second">
326             /// <para>The second.</para>
327             /// <para></para>
328             /// </param>
329             /// <returns>
330             /// <para>The uint</para>
331             /// <para></para>
332             /// </returns>
333             [MethodImpl(MethodImplOptions.AggressiveInlining)]
334             protected override uint Subtract(uint first, uint second) => first - second;
335
336             /// <summary>
337             /// <para>
338             /// Determines whether this instance first is to the left of second.
339             /// </para>
340             /// <para></para>
341             /// </summary>
342             /// <param name="first">
343             /// <para>The first.</para>
344             /// <para></para>
345             /// </param>
346             /// <param name="second">
347             /// <para>The second.</para>
348             /// <para></para>
349             /// </param>
350             /// <returns>
351             /// <para>The bool</para>
352             /// <para></para>
353             /// </returns>
354             [MethodImpl(MethodImplOptions.AggressiveInlining)]
355             protected override bool FirstIsToTheLeftOfSecond(uint first, uint second)
356             {
357                 ref var firstLink = ref Links[first];
358                 ref var secondLink = ref Links[second];
359                 return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
                    ↪  secondLink.Source, secondLink.Target);
360             }
361
362             /// <summary>
363             /// <para>
364             /// Determines whether this instance first is to the right of second.
365             /// </para>
366             /// <para></para>
367             /// </summary>
368             /// <param name="first">
369             /// <para>The first.</para>
370             /// <para></para>
371             /// </param>
372             /// <param name="second">
373             /// <para>The second.</para>
374             /// <para></para>
```

```
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [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);
        }

        /// <summary>
        /// <para>
        /// Gets the header reference.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A ref links header of uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref LinksHeader<uint> GetHeaderReference() => ref *Header;

        /// <summary>
        /// <para>
        /// Gets the link reference using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link of uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLink<uint> GetLinkReference(uint link) => ref Links[link];
    }
}
```

1.95 ./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
{
    /// <summary>
    /// <para>
    /// Represents the int 32 links sources recursionless size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="UInt32LinksRecursionlessSizeBalancedTreeMethodsBase"/>
    public unsafe class UInt32LinksSourcesRecursionlessSizeBalancedTreeMethods :
    ↪  UInt32LinksRecursionlessSizeBalancedTreeMethodsBase
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see
        ↪  cref="UInt32LinksSourcesRecursionlessSizeBalancedTreeMethods"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
```

```
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        public UInt32LinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<uint>
        ↪  constants, RawLink<uint>* links, LinksHeader<uint>* header) : base(constants, links,
        ↪  header) { }

        /// <summary>
        /// <para>
        /// Gets the left reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref uint GetLeftReference(uint node) => ref Links[node].LeftAsSource;

        /// <summary>
        /// <para>
        /// Gets the right reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref uint GetRightReference(uint node) => ref
        ↪  Links[node].RightAsSource;

        /// <summary>
        /// <para>
        /// Gets the left using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetLeft(uint node) => Links[node].LeftAsSource;

        /// <summary>
        /// <para>
        /// Gets the right using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetRight(uint node) => Links[node].RightAsSource;

        /// <summary>
```

```
        /// <para>
        /// Sets the left using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="left">
        /// <para>The left.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(uint node, uint left) => Links[node].LeftAsSource = left;

        /// <summary>
        /// <para>
        /// Sets the right using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="right">
        /// <para>The right.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(uint node, uint right) => Links[node].RightAsSource =
        ↪  right;

        /// <summary>
        /// <para>
        /// Gets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetSize(uint node) => Links[node].SizeAsSource;

        /// <summary>
        /// <para>
        /// Sets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="size">
        /// <para>The size.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(uint node, uint size) => Links[node].SizeAsSource = size;

        /// <summary>
        /// <para>
        /// Gets the tree root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The uint</para>
        /// <para></para>
        /// </returns>
```

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

        /// <summary>
        /// <para>
        /// Gets the base part value using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetBasePartValue(uint link) => Links[link].Source;

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="firstSource">
        /// <para>The first source.</para>
        /// <para></para>
        /// </param>
        /// <param name="firstTarget">
        /// <para>The first target.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondSource">
        /// <para>The second source.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondTarget">
        /// <para>The second target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(uint firstSource, uint firstTarget,
        ↪  uint secondSource, uint secondTarget)
            => firstSource < secondSource || (firstSource == secondSource && firstTarget <
            ↪  secondTarget);

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the right of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="firstSource">
        /// <para>The first source.</para>
        /// <para></para>
        /// </param>
        /// <param name="firstTarget">
        /// <para>The first target.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondSource">
        /// <para>The second source.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondTarget">
        /// <para>The second target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
```

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

        /// <summary>
        /// <para>
        /// Clears the node using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(uint node)
        {
            ref var link = ref Links[node];
            link.LeftAsSource = 0U;
            link.RightAsSource = 0U;
            link.SizeAsSource = 0U;
        }
    }
}
```

1.96 ./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
{
    /// <summary>
    /// <para>
    /// Represents the int 32 links sources size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="UInt32LinksSizeBalancedTreeMethodsBase"/>
    public unsafe class UInt32LinksSourcesSizeBalancedTreeMethods :
    ↪  UInt32LinksSizeBalancedTreeMethodsBase
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt32LinksSourcesSizeBalancedTreeMethods"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        public UInt32LinksSourcesSizeBalancedTreeMethods(LinksConstants<uint> constants,
        ↪  RawLink<uint>* links, LinksHeader<uint>* header) : base(constants, links, header) { }

        /// <summary>
        /// <para>
        /// Gets the left reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref uint</para>
        /// <para></para>
```

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

        /// <summary>
        /// <para>
        /// Gets the right reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref uint GetRightReference(uint node) => ref
        ↪  Links[node].RightAsSource;

        /// <summary>
        /// <para>
        /// Gets the left using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetLeft(uint node) => Links[node].LeftAsSource;

        /// <summary>
        /// <para>
        /// Gets the right using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetRight(uint node) => Links[node].RightAsSource;

        /// <summary>
        /// <para>
        /// Sets the left using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="left">
        /// <para>The left.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(uint node, uint left) => Links[node].LeftAsSource = left;

        /// <summary>
        /// <para>
        /// Sets the right using the specified node.
        /// </para>
        /// <para></para>
```

```
126         /// </summary>
127         /// <param name="node">
128         /// <para>The node.</para>
129         /// <para></para>
130         /// </param>
131         /// <param name="right">
132         /// <para>The right.</para>
133         /// <para></para>
134         /// </param>
135         [MethodImpl(MethodImplOptions.AggressiveInlining)]
136         protected override void SetRight(uint node, uint right) => Links[node].RightAsSource =
            ↪  right;
137
138         /// <summary>
139         /// <para>
140         /// Gets the size using the specified node.
141         /// </para>
142         /// <para></para>
143         /// </summary>
144         /// <param name="node">
145         /// <para>The node.</para>
146         /// <para></para>
147         /// </param>
148         /// <returns>
149         /// <para>The uint</para>
150         /// <para></para>
151         /// </returns>
152         [MethodImpl(MethodImplOptions.AggressiveInlining)]
153         protected override uint GetSize(uint node) => Links[node].SizeAsSource;
154
155         /// <summary>
156         /// <para>
157         /// Sets the size using the specified node.
158         /// </para>
159         /// <para></para>
160         /// </summary>
161         /// <param name="node">
162         /// <para>The node.</para>
163         /// <para></para>
164         /// </param>
165         /// <param name="size">
166         /// <para>The size.</para>
167         /// <para></para>
168         /// </param>
169         [MethodImpl(MethodImplOptions.AggressiveInlining)]
170         protected override void SetSize(uint node, uint size) => Links[node].SizeAsSource = size;
171
172         /// <summary>
173         /// <para>
174         /// Gets the tree root.
175         /// </para>
176         /// <para></para>
177         /// </summary>
178         /// <returns>
179         /// <para>The uint</para>
180         /// <para></para>
181         /// </returns>
182         [MethodImpl(MethodImplOptions.AggressiveInlining)]
183         protected override uint GetTreeRoot() => Header->RootAsSource;
184
185         /// <summary>
186         /// <para>
187         /// Gets the base part value using the specified link.
188         /// </para>
189         /// <para></para>
190         /// </summary>
191         /// <param name="link">
192         /// <para>The link.</para>
193         /// <para></para>
194         /// </param>
195         /// <returns>
196         /// <para>The uint</para>
197         /// <para></para>
198         /// </returns>
199         [MethodImpl(MethodImplOptions.AggressiveInlining)]
200         protected override uint GetBasePartValue(uint link) => Links[link].Source;
201
202         /// <summary>
```

```
203             /// <para>
204             /// Determines whether this instance first is to the left of second.
205             /// </para>
206             /// <para></para>
207             /// </summary>
208             /// <param name="firstSource">
209             /// <para>The first source.</para>
210             /// <para></para>
211             /// </param>
212             /// <param name="firstTarget">
213             /// <para>The first target.</para>
214             /// <para></para>
215             /// </param>
216             /// <param name="secondSource">
217             /// <para>The second source.</para>
218             /// <para></para>
219             /// </param>
220             /// <param name="secondTarget">
221             /// <para>The second target.</para>
222             /// <para></para>
223             /// </param>
224             /// <returns>
225             /// <para>The bool</para>
226             /// <para></para>
227             /// </returns>
228             [MethodImpl(MethodImplOptions.AggressiveInlining)]
229             protected override bool FirstIsToTheLeftOfSecond(uint firstSource, uint firstTarget,
                ↪  uint secondSource, uint secondTarget)
230                 => firstSource < secondSource || (firstSource == secondSource && firstTarget <
                    ↪  secondTarget);
231
232             /// <summary>
233             /// <para>
234             /// Determines whether this instance first is to the right of second.
235             /// </para>
236             /// <para></para>
237             /// </summary>
238             /// <param name="firstSource">
239             /// <para>The first source.</para>
240             /// <para></para>
241             /// </param>
242             /// <param name="firstTarget">
243             /// <para>The first target.</para>
244             /// <para></para>
245             /// </param>
246             /// <param name="secondSource">
247             /// <para>The second source.</para>
248             /// <para></para>
249             /// </param>
250             /// <param name="secondTarget">
251             /// <para>The second target.</para>
252             /// <para></para>
253             /// </param>
254             /// <returns>
255             /// <para>The bool</para>
256             /// <para></para>
257             /// </returns>
258             [MethodImpl(MethodImplOptions.AggressiveInlining)]
259             protected override bool FirstIsToTheRightOfSecond(uint firstSource, uint firstTarget,
                ↪  uint secondSource, uint secondTarget)
260                 => firstSource > secondSource || (firstSource == secondSource && firstTarget >
                    ↪  secondTarget);
261
262             /// <summary>
263             /// <para>
264             /// Clears the node using the specified node.
265             /// </para>
266             /// <para></para>
267             /// </summary>
268             /// <param name="node">
269             /// <para>The node.</para>
270             /// <para></para>
271             /// </param>
272             [MethodImpl(MethodImplOptions.AggressiveInlining)]
273             protected override void ClearNode(uint node)
274             {
275                 ref var link = ref Links[node];
```

```
            link.LeftAsSource = 0U;
            link.RightAsSource = 0U;
            link.SizeAsSource = 0U;
        }
    }
}
```

1.97 ./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
{
    /// <summary>
    /// <para>
    /// Represents the int 32 links targets recursionless size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="UInt32LinksRecursionlessSizeBalancedTreeMethodsBase"/>
    public unsafe class UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods :
    ↪  UInt32LinksRecursionlessSizeBalancedTreeMethodsBase
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see
        ↪  cref="UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        public UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<uint>
        ↪  constants, RawLink<uint>* links, LinksHeader<uint>* header) : base(constants, links,
        ↪  header) { }

        /// <summary>
        /// <para>
        /// Gets the left reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref uint GetLeftReference(uint node) => ref Links[node].LeftAsTarget;

        /// <summary>
        /// <para>
        /// Gets the right reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref uint</para>
        /// <para></para>
        /// </returns>
```

```
67          [MethodImpl(MethodImplOptions.AggressiveInlining)]
68          protected override ref uint GetRightReference(uint node) => ref
            ↪ Links[node].RightAsTarget;
69
70          /// <summary>
71          /// <para>
72          /// Gets the left using the specified node.
73          /// </para>
74          /// <para></para>
75          /// </summary>
76          /// <param name="node">
77          /// <para>The node.</para>
78          /// <para></para>
79          /// </param>
80          /// <returns>
81          /// <para>The uint</para>
82          /// <para></para>
83          /// </returns>
84          [MethodImpl(MethodImplOptions.AggressiveInlining)]
85          protected override uint GetLeft(uint node) => Links[node].LeftAsTarget;
86
87          /// <summary>
88          /// <para>
89          /// Gets the right using the specified node.
90          /// </para>
91          /// <para></para>
92          /// </summary>
93          /// <param name="node">
94          /// <para>The node.</para>
95          /// <para></para>
96          /// </param>
97          /// <returns>
98          /// <para>The uint</para>
99          /// <para></para>
100         /// </returns>
101         [MethodImpl(MethodImplOptions.AggressiveInlining)]
102         protected override uint GetRight(uint node) => Links[node].RightAsTarget;
103
104         /// <summary>
105         /// <para>
106         /// Sets the left using the specified node.
107         /// </para>
108         /// <para></para>
109         /// </summary>
110         /// <param name="node">
111         /// <para>The node.</para>
112         /// <para></para>
113         /// </param>
114         /// <param name="left">
115         /// <para>The left.</para>
116         /// <para></para>
117         /// </param>
118         [MethodImpl(MethodImplOptions.AggressiveInlining)]
119         protected override void SetLeft(uint node, uint left) => Links[node].LeftAsTarget = left;
120
121         /// <summary>
122         /// <para>
123         /// Sets the right using the specified node.
124         /// </para>
125         /// <para></para>
126         /// </summary>
127         /// <param name="node">
128         /// <para>The node.</para>
129         /// <para></para>
130         /// </param>
131         /// <param name="right">
132         /// <para>The right.</para>
133         /// <para></para>
134         /// </param>
135         [MethodImpl(MethodImplOptions.AggressiveInlining)]
136         protected override void SetRight(uint node, uint right) => Links[node].RightAsTarget =
            ↪ right;
137
138         /// <summary>
139         /// <para>
140         /// Gets the size using the specified node.
141         /// </para>
142         /// <para></para>
```

```
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetSize(uint node) => Links[node].SizeAsTarget;

        /// <summary>
        /// <para>
        /// Sets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="size">
        /// <para>The size.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(uint node, uint size) => Links[node].SizeAsTarget = size;

        /// <summary>
        /// <para>
        /// Gets the tree root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetTreeRoot() => Header->RootAsTarget;

        /// <summary>
        /// <para>
        /// Gets the base part value using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetBasePartValue(uint link) => Links[link].Target;

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="firstSource">
        /// <para>The first source.</para>
        /// <para></para>
        /// </param>
        /// <param name="firstTarget">
        /// <para>The first target.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondSource">
        /// <para>The second source.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondTarget">
```

```
        /// <para>The second target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(uint firstSource, uint firstTarget,
        ↪  uint secondSource, uint secondTarget)
            => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
            ↪  secondSource);

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the right of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="firstSource">
        /// <para>The first source.</para>
        /// <para></para>
        /// </param>
        /// <param name="firstTarget">
        /// <para>The first target.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondSource">
        /// <para>The second source.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondTarget">
        /// <para>The second target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(uint firstSource, uint firstTarget,
        ↪  uint secondSource, uint secondTarget)
            => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
            ↪  secondSource);

        /// <summary>
        /// <para>
        /// Clears the node using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(uint node)
        {
            ref var link = ref Links[node];
            link.LeftAsTarget = 0U;
            link.RightAsTarget = 0U;
            link.SizeAsTarget = 0U;
        }
    }
}
```

1.98 ./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
{
    /// <summary>
    /// <para>
    /// Represents the int 32 links targets size balanced tree methods.
    /// </para>
    /// <para></para>
```

```
    /// </summary>
    /// <seealso cref="UInt32LinksSizeBalancedTreeMethodsBase"/>
    public unsafe class UInt32LinksTargetsSizeBalancedTreeMethods :
    ↪ UInt32LinksSizeBalancedTreeMethodsBase
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt32LinksTargetsSizeBalancedTreeMethods"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        public UInt32LinksTargetsSizeBalancedTreeMethods(LinksConstants<uint> constants,
        ↪ RawLink<uint>* links, LinksHeader<uint>* header) : base(constants, links, header) { }

        /// <summary>
        /// <para>
        /// Gets the left reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref uint GetLeftReference(uint node) => ref Links[node].LeftAsTarget;

        /// <summary>
        /// <para>
        /// Gets the right reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref uint GetRightReference(uint node) => ref
        ↪ Links[node].RightAsTarget;

        /// <summary>
        /// <para>
        /// Gets the left using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetLeft(uint node) => Links[node].LeftAsTarget;

```

```
        /// <summary>
        /// <para>
        /// Gets the right using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetRight(uint node) => Links[node].RightAsTarget;

        /// <summary>
        /// <para>
        /// Sets the left using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="left">
        /// <para>The left.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(uint node, uint left) => Links[node].LeftAsTarget = left;

        /// <summary>
        /// <para>
        /// Sets the right using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="right">
        /// <para>The right.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(uint node, uint right) => Links[node].RightAsTarget =
        ↪  right;

        /// <summary>
        /// <para>
        /// Gets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetSize(uint node) => Links[node].SizeAsTarget;

        /// <summary>
        /// <para>
        /// Sets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
```

```
        /// </param>
        /// <param name="size">
        /// <para>The size.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(uint node, uint size) => Links[node].SizeAsTarget = size;

        /// <summary>
        /// <para>
        /// Gets the tree root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetTreeRoot() => Header->RootAsTarget;

        /// <summary>
        /// <para>
        /// Gets the base part value using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetBasePartValue(uint link) => Links[link].Target;

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="firstSource">
        /// <para>The first source.</para>
        /// <para></para>
        /// </param>
        /// <param name="firstTarget">
        /// <para>The first target.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondSource">
        /// <para>The second source.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondTarget">
        /// <para>The second target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(uint firstSource, uint firstTarget,
        ↪  uint secondSource, uint secondTarget)
            => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
            ↪  secondSource);

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the right of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="firstSource">
        /// <para>The first source.</para>
```

```
240         /// <para></para>
241         /// </param>
242         /// <param name="firstTarget">
243         /// <para>The first target.</para>
244         /// <para></para>
245         /// </param>
246         /// <param name="secondSource">
247         /// <para>The second source.</para>
248         /// <para></para>
249         /// </param>
250         /// <param name="secondTarget">
251         /// <para>The second target.</para>
252         /// <para></para>
253         /// </param>
254         /// <returns>
255         /// <para>The bool</para>
256         /// <para></para>
257         /// </returns>
258         [MethodImpl(MethodImplOptions.AggressiveInlining)]
259         protected override bool FirstIsToTheRightOfSecond(uint firstSource, uint firstTarget,
            ↪ uint secondSource, uint secondTarget)
260             => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
                ↪ secondSource);
261 
262         /// <summary>
263         /// <para>
264         /// Clears the node using the specified node.
265         /// </para>
266         /// <para></para>
267         /// </summary>
268         /// <param name="node">
269         /// <para>The node.</para>
270         /// <para></para>
271         /// </param>
272         [MethodImpl(MethodImplOptions.AggressiveInlining)]
273         protected override void ClearNode(uint node)
274         {
275             ref var link = ref Links[node];
276             link.LeftAsTarget = 0U;
277             link.RightAsTarget = 0U;
278             link.SizeAsTarget = 0U;
279         }
280     }
281 }
```

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

```
1  using System;
2  using System.Runtime.CompilerServices;
3  using Platform.Memory;
4  using Platform.Singletons;
5  using Platform.Data.Doublets.Memory.United.Generic;
6 
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8 
9  namespace Platform.Data.Doublets.Memory.United.Specific
10 {
11     /// <summary>
12     /// <para>Represents a low-level implementation of direct access to resizable memory, for
        ↪ organizing the storage of links with addresses represented as <see cref="uint" />.</para>
13     /// <para>Представляет низкоуровневую реализация прямого доступа к памяти с переменным
        ↪ размером, для организации хранения связей с адресами представленными в виде <see
        ↪ cref="uint"/>.</para>
14     /// </summary>
15     public unsafe class UInt32UnitedMemoryLinks : UnitedMemoryLinksBase<uint>
16     {
17         private readonly Func<ILinksTreeMethods<uint>> _createSourceTreeMethods;
18         private readonly Func<ILinksTreeMethods<uint>> _createTargetTreeMethods;
19         private LinksHeader<uint>* _header;
20         private RawLink<uint>* _links;
21 
22         /// <summary>
23         /// <para>
24         /// Initializes a new <see cref="UInt32UnitedMemoryLinks"/> instance.
25         /// </para>
26         /// <para></para>
27         /// </summary>
28         /// <param name="address">
29         /// <para>A address.</para>
30         /// <para></para>
```

```
31          /// </param>
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          public UInt32UnitedMemoryLinks(string address) : this(address, DefaultLinksSizeStep) { }
34
35          /// <summary>
36          /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
            ↪  минимальным шагом расширения базы данных.
37          /// </summary>
38          /// <param name="address">Полный пусть к файлу базы данных.</param>
39          /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
            ↪  байтах.</param>
40          [MethodImpl(MethodImplOptions.AggressiveInlining)]
41          public UInt32UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
            ↪  FileMappedResizableDirectMemory(address, memoryReservationStep),
            ↪  memoryReservationStep) { }
42
43          /// <summary>
44          /// <para>
45          /// Initializes a new <see cref="UInt32UnitedMemoryLinks"/> instance.
46          /// </para>
47          /// <para></para>
48          /// </summary>
49          /// <param name="memory">
50          /// <para>A memory.</para>
51          /// <para></para>
52          /// </param>
53          [MethodImpl(MethodImplOptions.AggressiveInlining)]
54          public UInt32UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
            ↪  DefaultLinksSizeStep) { }
55
56          /// <summary>
57          /// <para>
58          /// Initializes a new <see cref="UInt32UnitedMemoryLinks"/> instance.
59          /// </para>
60          /// <para></para>
61          /// </summary>
62          /// <param name="memory">
63          /// <para>A memory.</para>
64          /// <para></para>
65          /// </param>
66          /// <param name="memoryReservationStep">
67          /// <para>A memory reservation step.</para>
68          /// <para></para>
69          /// </param>
70          [MethodImpl(MethodImplOptions.AggressiveInlining)]
71          public UInt32UnitedMemoryLinks(IResizableDirectMemory memory, long
            ↪  memoryReservationStep) : this(memory, memoryReservationStep,
            ↪  Default<LinksConstants<uint>>.Instance, IndexTreeType.Default) { }
72
73          /// <summary>
74          /// <para>
75          /// Initializes a new <see cref="UInt32UnitedMemoryLinks"/> instance.
76          /// </para>
77          /// <para></para>
78          /// </summary>
79          /// <param name="memory">
80          /// <para>A memory.</para>
81          /// <para></para>
82          /// </param>
83          /// <param name="memoryReservationStep">
84          /// <para>A memory reservation step.</para>
85          /// <para></para>
86          /// </param>
87          /// <param name="constants">
88          /// <para>A constants.</para>
89          /// <para></para>
90          /// </param>
91          /// <param name="indexTreeType">
92          /// <para>A index tree type.</para>
93          /// <para></para>
94          /// </param>
95          [MethodImpl(MethodImplOptions.AggressiveInlining)]
96          public UInt32UnitedMemoryLinks(IResizableDirectMemory memory, long
            ↪  memoryReservationStep, LinksConstants<uint> constants, IndexTreeType indexTreeType)
            ↪  : base(memory, memoryReservationStep, constants)
97          {
98              if (indexTreeType == IndexTreeType.SizeBalancedTree)
```

```
99              {
100                 _createSourceTreeMethods = () => new
                    ↪ UInt32LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
101                 _createTargetTreeMethods = () => new
                    ↪ UInt32LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
102             }
103             else
104             {
105                 _createSourceTreeMethods = () => new
                    ↪ UInt32LinksSourcesRecursionlessSizeBalancedTreeMethods(Constants, _links,
                    ↪ _header);
106                 _createTargetTreeMethods = () => new
                    ↪ UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods(Constants, _links,
                    ↪ _header);
107             }
108             Init(memory, memoryReservationStep);
109         }
110
111         /// <summary>
112         /// <para>
113         /// Sets the pointers using the specified memory.
114         /// </para>
115         /// <para></para>
116         /// </summary>
117         /// <param name="memory">
118         /// <para>The memory.</para>
119         /// <para></para>
120         /// </param>
121         [MethodImpl(MethodImplOptions.AggressiveInlining)]
122         protected override void SetPointers(IResizableDirectMemory memory)
123         {
124             _header = (LinksHeader<uint>*)memory.Pointer;
125             _links = (RawLink<uint>*)memory.Pointer;
126             SourcesTreeMethods = _createSourceTreeMethods();
127             TargetsTreeMethods = _createTargetTreeMethods();
128             UnusedLinksListMethods = new UInt32UnusedLinksListMethods(_links, _header);
129         }
130
131         /// <summary>
132         /// <para>
133         /// Resets the pointers.
134         /// </para>
135         /// <para></para>
136         /// </summary>
137         [MethodImpl(MethodImplOptions.AggressiveInlining)]
138         protected override void ResetPointers()
139         {
140             base.ResetPointers();
141             _links = null;
142             _header = null;
143         }
144
145         /// <summary>
146         /// <para>
147         /// Gets the header reference.
148         /// </para>
149         /// <para></para>
150         /// </summary>
151         /// <returns>
152         /// <para>A ref links header of uint</para>
153         /// <para></para>
154         /// </returns>
155         [MethodImpl(MethodImplOptions.AggressiveInlining)]
156         protected override ref LinksHeader<uint> GetHeaderReference() => ref *_header;
157
158         /// <summary>
159         /// <para>
160         /// Gets the link reference using the specified link index.
161         /// </para>
162         /// <para></para>
163         /// </summary>
164         /// <param name="linkIndex">
165         /// <para>The link index.</para>
166         /// <para></para>
167         /// </param>
168         /// <returns>
169         /// <para>A ref raw link of uint</para>
170         /// <para></para>
```

```
171         /// </returns>
172         [MethodImpl(MethodImplOptions.AggressiveInlining)]
173         protected override ref RawLink<uint> GetLinkReference(uint linkIndex) => ref
            ↪  _links[linkIndex];
174
175         /// <summary>
176         /// <para>
177         /// Determines whether this instance are equal.
178         /// </para>
179         /// <para></para>
180         /// </summary>
181         /// <param name="first">
182         /// <para>The first.</para>
183         /// <para></para>
184         /// </param>
185         /// <param name="second">
186         /// <para>The second.</para>
187         /// <para></para>
188         /// </param>
189         /// <returns>
190         /// <para>The bool</para>
191         /// <para></para>
192         /// </returns>
193         [MethodImpl(MethodImplOptions.AggressiveInlining)]
194         protected override bool AreEqual(uint first, uint second) => first == second;
195
196         /// <summary>
197         /// <para>
198         /// Determines whether this instance less than.
199         /// </para>
200         /// <para></para>
201         /// </summary>
202         /// <param name="first">
203         /// <para>The first.</para>
204         /// <para></para>
205         /// </param>
206         /// <param name="second">
207         /// <para>The second.</para>
208         /// <para></para>
209         /// </param>
210         /// <returns>
211         /// <para>The bool</para>
212         /// <para></para>
213         /// </returns>
214         [MethodImpl(MethodImplOptions.AggressiveInlining)]
215         protected override bool LessThan(uint first, uint second) => first < second;
216
217         /// <summary>
218         /// <para>
219         /// Determines whether this instance less or equal than.
220         /// </para>
221         /// <para></para>
222         /// </summary>
223         /// <param name="first">
224         /// <para>The first.</para>
225         /// <para></para>
226         /// </param>
227         /// <param name="second">
228         /// <para>The second.</para>
229         /// <para></para>
230         /// </param>
231         /// <returns>
232         /// <para>The bool</para>
233         /// <para></para>
234         /// </returns>
235         [MethodImpl(MethodImplOptions.AggressiveInlining)]
236         protected override bool LessOrEqualThan(uint first, uint second) => first <= second;
237
238         /// <summary>
239         /// <para>
240         /// Determines whether this instance greater than.
241         /// </para>
242         /// <para></para>
243         /// </summary>
244         /// <param name="first">
245         /// <para>The first.</para>
246         /// <para></para>
247         /// </param>
```

```
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThan(uint first, uint second) => first > second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater or equal than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterOrEqualThan(uint first, uint second) => first >= second;

        /// <summary>
        /// <para>
        /// Gets the zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetZero() => 0U;

        /// <summary>
        /// <para>
        /// Gets the one.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetOne() => 1U;

        /// <summary>
        /// <para>
        /// Converts the to int 64 using the specified value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The long</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override long ConvertToInt64(uint value) => (long)value;

        /// <summary>
        /// <para>
        /// Converts the to address using the specified value.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint ConvertToAddress(long value) => (uint)value;

        /// <summary>
        /// <para>
        /// Adds the first.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint Add(uint first, uint second) => first + second;

        /// <summary>
        /// <para>
        /// Subtracts the first.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint Subtract(uint first, uint second) => first - second;

        /// <summary>
        /// <para>
        /// Increments the link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint Increment(uint link) => ++link;

        /// <summary>
        /// <para>
        /// Decrements the link.
        /// </para>
        /// <para></para>
```

```
404         /// </summary>
405         /// <param name="link">
406         /// <para>The link.</para>
407         /// <para></para>
408         /// </param>
409         /// <returns>
410         /// <para>The uint</para>
411         /// <para></para>
412         /// </returns>
413         [MethodImpl(MethodImplOptions.AggressiveInlining)]
414         protected override uint Decrement(uint link) => --link;
415     }
416 }
```

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

```
1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.Memory.United.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.United.Specific
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the int 32 unused links list methods.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     /// <seealso cref="UnusedLinksListMethods{uint}"/>
15     public unsafe class UInt32UnusedLinksListMethods : UnusedLinksListMethods<uint>
16     {
17         private readonly RawLink<uint>* _links;
18         private readonly LinksHeader<uint>* _header;
19
20         /// <summary>
21         /// <para>
22         /// Initializes a new <see cref="UInt32UnusedLinksListMethods"/> instance.
23         /// </para>
24         /// <para></para>
25         /// </summary>
26         /// <param name="links">
27         /// <para>A links.</para>
28         /// <para></para>
29         /// </param>
30         /// <param name="header">
31         /// <para>A header.</para>
32         /// <para></para>
33         /// </param>
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public UInt32UnusedLinksListMethods(RawLink<uint>* links, LinksHeader<uint>* header)
36             : base((byte*)links, (byte*)header)
37         {
38             _links = links;
39             _header = header;
40         }
41
42         /// <summary>
43         /// <para>
44         /// Gets the link reference using the specified link.
45         /// </para>
46         /// <para></para>
47         /// </summary>
48         /// <param name="link">
49         /// <para>The link.</para>
50         /// <para></para>
51         /// </param>
52         /// <returns>
53         /// <para>A ref raw link of uint</para>
54         /// <para></para>
55         /// </returns>
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         protected override ref RawLink<uint> GetLinkReference(uint link) => ref _links[link];
58
59         /// <summary>
60         /// <para>
61         /// Gets the header reference.
62         /// </para>
63         /// <para></para>
64         /// </summary>
```

```
        /// <returns>
        /// <para>A ref links header of uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref LinksHeader<uint> GetHeaderReference() => ref *_header;
    }
}
```

1.101 ./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
{
    /// <summary>
    /// <para>
    /// Represents the int 64 links avl balanced tree methods base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksAvlBalancedTreeMethodsBase{ulong}"/>
    public unsafe abstract class UInt64LinksAvlBalancedTreeMethodsBase :
    ↪ LinksAvlBalancedTreeMethodsBase<ulong>
    {
        /// <summary>
        /// <para>
        /// The links.
        /// </para>
        /// <para></para>
        /// </summary>
        protected new readonly RawLink<ulong>* Links;
        /// <summary>
        /// <para>
        /// The header.
        /// </para>
        /// <para></para>
        /// </summary>
        protected new readonly LinksHeader<ulong>* Header;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt64LinksAvlBalancedTreeMethodsBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        protected UInt64LinksAvlBalancedTreeMethodsBase(LinksConstants<ulong> constants,
        ↪ RawLink<ulong>* links, LinksHeader<ulong>* header)
            : base(constants, (byte*)links, (byte*)header)
        {
            Links = links;
            Header = header;
        }

        /// <summary>
        /// <para>
        /// Gets the zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetZero() => OUL;

        /// <summary>
        /// <para>
        /// Determines whether this instance equal to zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool EqualToZero(ulong value) => value == OUL;

        /// <summary>
        /// <para>
        /// Determines whether this instance are equal.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool AreEqual(ulong first, ulong second) => first == second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThanZero(ulong value) => value > OUL;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThan(ulong first, ulong second) => first > second;
```

```
146
147         /// <summary>
148         /// <para>
149         /// Determines whether this instance greater or equal than.
150         /// </para>
151         /// <para></para>
152         /// </summary>
153         /// <param name="first">
154         /// <para>The first.</para>
155         /// <para></para>
156         /// </param>
157         /// <param name="second">
158         /// <para>The second.</para>
159         /// <para></para>
160         /// </param>
161         /// <returns>
162         /// <para>The bool</para>
163         /// <para></para>
164         /// </returns>
165         [MethodImpl(MethodImplOptions.AggressiveInlining)]
166         protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
167
168         /// <summary>
169         /// <para>
170         /// Determines whether this instance greater or equal than zero.
171         /// </para>
172         /// <para></para>
173         /// </summary>
174         /// <param name="value">
175         /// <para>The value.</para>
176         /// <para></para>
177         /// </param>
178         /// <returns>
179         /// <para>The bool</para>
180         /// <para></para>
181         /// </returns>
182         [MethodImpl(MethodImplOptions.AggressiveInlining)]
183         protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
             ↪  always true for ulong
184
185         /// <summary>
186         /// <para>
187         /// Determines whether this instance less or equal than zero.
188         /// </para>
189         /// <para></para>
190         /// </summary>
191         /// <param name="value">
192         /// <para>The value.</para>
193         /// <para></para>
194         /// </param>
195         /// <returns>
196         /// <para>The bool</para>
197         /// <para></para>
198         /// </returns>
199         [MethodImpl(MethodImplOptions.AggressiveInlining)]
200         protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
             ↪  always >= 0 for ulong
201
202         /// <summary>
203         /// <para>
204         /// Determines whether this instance less or equal than.
205         /// </para>
206         /// <para></para>
207         /// </summary>
208         /// <param name="first">
209         /// <para>The first.</para>
210         /// <para></para>
211         /// </param>
212         /// <param name="second">
213         /// <para>The second.</para>
214         /// <para></para>
215         /// </param>
216         /// <returns>
217         /// <para>The bool</para>
218         /// <para></para>
219         /// </returns>
220         [MethodImpl(MethodImplOptions.AggressiveInlining)]
221         protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
```

```
222
223         /// <summary>
224         /// <para>
225         /// Determines whether this instance less than zero.
226         /// </para>
227         /// <para></para>
228         /// </summary>
229         /// <param name="value">
230         /// <para>The value.</para>
231         /// <para></para>
232         /// </param>
233         /// <returns>
234         /// <para>The bool</para>
235         /// <para></para>
236         /// </returns>
237         [MethodImpl(MethodImplOptions.AggressiveInlining)]
238         protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
            ↪  for ulong
239
240         /// <summary>
241         /// <para>
242         /// Determines whether this instance less than.
243         /// </para>
244         /// <para></para>
245         /// </summary>
246         /// <param name="first">
247         /// <para>The first.</para>
248         /// <para></para>
249         /// </param>
250         /// <param name="second">
251         /// <para>The second.</para>
252         /// <para></para>
253         /// </param>
254         /// <returns>
255         /// <para>The bool</para>
256         /// <para></para>
257         /// </returns>
258         [MethodImpl(MethodImplOptions.AggressiveInlining)]
259         protected override bool LessThan(ulong first, ulong second) => first < second;
260
261         /// <summary>
262         /// <para>
263         /// Increments the value.
264         /// </para>
265         /// <para></para>
266         /// </summary>
267         /// <param name="value">
268         /// <para>The value.</para>
269         /// <para></para>
270         /// </param>
271         /// <returns>
272         /// <para>The ulong</para>
273         /// <para></para>
274         /// </returns>
275         [MethodImpl(MethodImplOptions.AggressiveInlining)]
276         protected override ulong Increment(ulong value) => ++value;
277
278         /// <summary>
279         /// <para>
280         /// Decrements the value.
281         /// </para>
282         /// <para></para>
283         /// </summary>
284         /// <param name="value">
285         /// <para>The value.</para>
286         /// <para></para>
287         /// </param>
288         /// <returns>
289         /// <para>The ulong</para>
290         /// <para></para>
291         /// </returns>
292         [MethodImpl(MethodImplOptions.AggressiveInlining)]
293         protected override ulong Decrement(ulong value) => --value;
294
295         /// <summary>
296         /// <para>
297         /// Adds the first.
298         /// </para>
```

```
299             /// <para></para>
300             /// </summary>
301             /// <param name="first">
302             /// <para>The first.</para>
303             /// <para></para>
304             /// </param>
305             /// <param name="second">
306             /// <para>The second.</para>
307             /// <para></para>
308             /// </param>
309             /// <returns>
310             /// <para>The ulong</para>
311             /// <para></para>
312             /// </returns>
313             [MethodImpl(MethodImplOptions.AggressiveInlining)]
314             protected override ulong Add(ulong first, ulong second) => first + second;
315
316             /// <summary>
317             /// <para>
318             /// Subtracts the first.
319             /// </para>
320             /// <para></para>
321             /// </summary>
322             /// <param name="first">
323             /// <para>The first.</para>
324             /// <para></para>
325             /// </param>
326             /// <param name="second">
327             /// <para>The second.</para>
328             /// <para></para>
329             /// </param>
330             /// <returns>
331             /// <para>The ulong</para>
332             /// <para></para>
333             /// </returns>
334             [MethodImpl(MethodImplOptions.AggressiveInlining)]
335             protected override ulong Subtract(ulong first, ulong second) => first - second;
336
337             /// <summary>
338             /// <para>
339             /// Determines whether this instance first is to the left of second.
340             /// </para>
341             /// <para></para>
342             /// </summary>
343             /// <param name="first">
344             /// <para>The first.</para>
345             /// <para></para>
346             /// </param>
347             /// <param name="second">
348             /// <para>The second.</para>
349             /// <para></para>
350             /// </param>
351             /// <returns>
352             /// <para>The bool</para>
353             /// <para></para>
354             /// </returns>
355             [MethodImpl(MethodImplOptions.AggressiveInlining)]
356             protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
357             {
358                 ref var firstLink = ref Links[first];
359                 ref var secondLink = ref Links[second];
360                 return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
                    ↪  secondLink.Source, secondLink.Target);
361             }
362
363             /// <summary>
364             /// <para>
365             /// Determines whether this instance first is to the right of second.
366             /// </para>
367             /// <para></para>
368             /// </summary>
369             /// <param name="first">
370             /// <para>The first.</para>
371             /// <para></para>
372             /// </param>
373             /// <param name="second">
374             /// <para>The second.</para>
375             /// <para></para>
```

```
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [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);
        }

        /// <summary>
        /// <para>
        /// Gets the size value using the specified value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetSizeValue(ulong value) => (value & 4294967264UL) >> 5;

        /// <summary>
        /// <para>
        /// Sets the size value using the specified stored value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="storedValue">
        /// <para>The stored value.</para>
        /// <para></para>
        /// </param>
        /// <param name="size">
        /// <para>The size.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSizeValue(ref ulong storedValue, ulong size) => storedValue =
        ↪  storedValue & 31UL | (size & 134217727UL) << 5;

        /// <summary>
        /// <para>
        /// Determines whether this instance get left is child value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetLeftIsChildValue(ulong value) => (value & 16UL) >> 4 == 1UL;

        /// <summary>
        /// <para>
        /// Sets the left is child value using the specified stored value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="storedValue">
        /// <para>The stored value.</para>
        /// <para></para>
        /// </param>
        /// <param name="value">
        /// <para>The value.</para>
```

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

        /// <summary>
        /// <para>
        /// Determines whether this instance get right is child value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetRightIsChildValue(ulong value) => (value & 8UL) >> 3 == 1UL;

        /// <summary>
        /// <para>
        /// Sets the right is child value using the specified stored value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="storedValue">
        /// <para>The stored value.</para>
        /// <para></para>
        /// </param>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRightIsChildValue(ref ulong storedValue, bool value) =>
        ↪  storedValue = storedValue & 4294967287UL | (As<bool, byte>(ref value) & 1UL) << 3;

        /// <summary>
        /// <para>
        /// Gets the balance value using the specified value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The sbyte</para>
        /// <para></para>
        /// </returns>
        [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

        /// <summary>
        /// <para>
        /// Sets the balance value using the specified stored value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="storedValue">
        /// <para>The stored value.</para>
        /// <para></para>
        /// </param>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetBalanceValue(ref ulong storedValue, sbyte value) =>
        ↪  storedValue = unchecked(storedValue & 4294967288UL | (ulong)((byte)value >> 5 & 4 |
        ↪  value & 3) & 7UL);
```

```
524
525         /// <summary>
526         /// <para>
527         /// Gets the header reference.
528         /// </para>
529         /// <para></para>
530         /// </summary>
531         /// <returns>
532         /// <para>A ref links header of ulong</para>
533         /// <para></para>
534         /// </returns>
535         [MethodImpl(MethodImplOptions.AggressiveInlining)]
536         protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
537
538         /// <summary>
539         /// <para>
540         /// Gets the link reference using the specified link.
541         /// </para>
542         /// <para></para>
543         /// </summary>
544         /// <param name="link">
545         /// <para>The link.</para>
546         /// <para></para>
547         /// </param>
548         /// <returns>
549         /// <para>A ref raw link of ulong</para>
550         /// <para></para>
551         /// </returns>
552         [MethodImpl(MethodImplOptions.AggressiveInlining)]
553         protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
554     }
555 }
```

1.102 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksRecursionlessSizeBalancedTreeMeth

```
1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.Memory.United.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.United.Specific
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the int 64 links recursionless size balanced tree methods base.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     /// <seealso cref="LinksRecursionlessSizeBalancedTreeMethodsBase{ulong}"/>
15     public unsafe abstract class UInt64LinksRecursionlessSizeBalancedTreeMethodsBase :
       ↪  LinksRecursionlessSizeBalancedTreeMethodsBase<ulong>
16     {
17         /// <summary>
18         /// <para>
19         /// The links.
20         /// </para>
21         /// <para></para>
22         /// </summary>
23         protected new readonly RawLink<ulong>* Links;
24         /// <summary>
25         /// <para>
26         /// The header.
27         /// </para>
28         /// <para></para>
29         /// </summary>
30         protected new readonly LinksHeader<ulong>* Header;
31
32         /// <summary>
33         /// <para>
34         /// Initializes a new <see cref="UInt64LinksRecursionlessSizeBalancedTreeMethodsBase"/>
           ↪  instance.
35         /// </para>
36         /// <para></para>
37         /// </summary>
38         /// <param name="constants">
39         /// <para>A constants.</para>
40         /// <para></para>
41         /// </param>
42         /// <param name="links">
```

```
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        protected UInt64LinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<ulong>
        ↪  constants, RawLink<ulong>* links, LinksHeader<ulong>* header)
            : base(constants, (byte*)links, (byte*)header)
        {
            Links = links;
            Header = header;
        }

        /// <summary>
        /// <para>
        /// Gets the zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetZero() => 0UL;

        /// <summary>
        /// <para>
        /// Determines whether this instance equal to zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool EqualToZero(ulong value) => value == 0UL;

        /// <summary>
        /// <para>
        /// Determines whether this instance are equal.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool AreEqual(ulong first, ulong second) => first == second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
```

```
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThanZero(ulong value) => value > 0UL;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThan(ulong first, ulong second) => first > second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater or equal than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater or equal than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
        ↪  always true for ulong

        /// <summary>
        /// <para>
        /// Determines whether this instance less or equal than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
```

```
197         /// </returns>
198         [MethodImpl(MethodImplOptions.AggressiveInlining)]
199         protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
            ↪  always >= 0 for ulong
200
201         /// <summary>
202         /// <para>
203         /// Determines whether this instance less or equal than.
204         /// </para>
205         /// <para></para>
206         /// </summary>
207         /// <param name="first">
208         /// <para>The first.</para>
209         /// <para></para>
210         /// </param>
211         /// <param name="second">
212         /// <para>The second.</para>
213         /// <para></para>
214         /// </param>
215         /// <returns>
216         /// <para>The bool</para>
217         /// <para></para>
218         /// </returns>
219         [MethodImpl(MethodImplOptions.AggressiveInlining)]
220         protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
221
222         /// <summary>
223         /// <para>
224         /// Determines whether this instance less than zero.
225         /// </para>
226         /// <para></para>
227         /// </summary>
228         /// <param name="value">
229         /// <para>The value.</para>
230         /// <para></para>
231         /// </param>
232         /// <returns>
233         /// <para>The bool</para>
234         /// <para></para>
235         /// </returns>
236         [MethodImpl(MethodImplOptions.AggressiveInlining)]
237         protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
            ↪  for ulong
238
239         /// <summary>
240         /// <para>
241         /// Determines whether this instance less than.
242         /// </para>
243         /// <para></para>
244         /// </summary>
245         /// <param name="first">
246         /// <para>The first.</para>
247         /// <para></para>
248         /// </param>
249         /// <param name="second">
250         /// <para>The second.</para>
251         /// <para></para>
252         /// </param>
253         /// <returns>
254         /// <para>The bool</para>
255         /// <para></para>
256         /// </returns>
257         [MethodImpl(MethodImplOptions.AggressiveInlining)]
258         protected override bool LessThan(ulong first, ulong second) => first < second;
259
260         /// <summary>
261         /// <para>
262         /// Increments the value.
263         /// </para>
264         /// <para></para>
265         /// </summary>
266         /// <param name="value">
267         /// <para>The value.</para>
268         /// <para></para>
269         /// </param>
270         /// <returns>
271         /// <para>The ulong</para>
272         /// <para></para>
```

```
273             /// </returns>
274             [MethodImpl(MethodImplOptions.AggressiveInlining)]
275             protected override ulong Increment(ulong value) => ++value;
276
277             /// <summary>
278             /// <para>
279             /// Decrements the value.
280             /// </para>
281             /// <para></para>
282             /// </summary>
283             /// <param name="value">
284             /// <para>The value.</para>
285             /// <para></para>
286             /// </param>
287             /// <returns>
288             /// <para>The ulong</para>
289             /// <para></para>
290             /// </returns>
291             [MethodImpl(MethodImplOptions.AggressiveInlining)]
292             protected override ulong Decrement(ulong value) => --value;
293
294             /// <summary>
295             /// <para>
296             /// Adds the first.
297             /// </para>
298             /// <para></para>
299             /// </summary>
300             /// <param name="first">
301             /// <para>The first.</para>
302             /// <para></para>
303             /// </param>
304             /// <param name="second">
305             /// <para>The second.</para>
306             /// <para></para>
307             /// </param>
308             /// <returns>
309             /// <para>The ulong</para>
310             /// <para></para>
311             /// </returns>
312             [MethodImpl(MethodImplOptions.AggressiveInlining)]
313             protected override ulong Add(ulong first, ulong second) => first + second;
314
315             /// <summary>
316             /// <para>
317             /// Subtracts the first.
318             /// </para>
319             /// <para></para>
320             /// </summary>
321             /// <param name="first">
322             /// <para>The first.</para>
323             /// <para></para>
324             /// </param>
325             /// <param name="second">
326             /// <para>The second.</para>
327             /// <para></para>
328             /// </param>
329             /// <returns>
330             /// <para>The ulong</para>
331             /// <para></para>
332             /// </returns>
333             [MethodImpl(MethodImplOptions.AggressiveInlining)]
334             protected override ulong Subtract(ulong first, ulong second) => first - second;
335
336             /// <summary>
337             /// <para>
338             /// Determines whether this instance first is to the left of second.
339             /// </para>
340             /// <para></para>
341             /// </summary>
342             /// <param name="first">
343             /// <para>The first.</para>
344             /// <para></para>
345             /// </param>
346             /// <param name="second">
347             /// <para>The second.</para>
348             /// <para></para>
349             /// </param>
350             /// <returns>
```

```
351         /// <para>The bool</para>
352         /// <para></para>
353         /// </returns>
354         [MethodImpl(MethodImplOptions.AggressiveInlining)]
355         protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
356         {
357             ref var firstLink = ref Links[first];
358             ref var secondLink = ref Links[second];
359             return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
360         }
361 
362         /// <summary>
363         /// <para>
364         /// Determines whether this instance first is to the right of second.
365         /// </para>
366         /// <para></para>
367         /// </summary>
368         /// <param name="first">
369         /// <para>The first.</para>
370         /// <para></para>
371         /// </param>
372         /// <param name="second">
373         /// <para>The second.</para>
374         /// <para></para>
375         /// </param>
376         /// <returns>
377         /// <para>The bool</para>
378         /// <para></para>
379         /// </returns>
380         [MethodImpl(MethodImplOptions.AggressiveInlining)]
381         protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
382         {
383             ref var firstLink = ref Links[first];
384             ref var secondLink = ref Links[second];
385             return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
386         }
387 
388         /// <summary>
389         /// <para>
390         /// Gets the header reference.
391         /// </para>
392         /// <para></para>
393         /// </summary>
394         /// <returns>
395         /// <para>A ref links header of ulong</para>
396         /// <para></para>
397         /// </returns>
398         [MethodImpl(MethodImplOptions.AggressiveInlining)]
399         protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
400 
401         /// <summary>
402         /// <para>
403         /// Gets the link reference using the specified link.
404         /// </para>
405         /// <para></para>
406         /// </summary>
407         /// <param name="link">
408         /// <para>The link.</para>
409         /// <para></para>
410         /// </param>
411         /// <returns>
412         /// <para>A ref raw link of ulong</para>
413         /// <para></para>
414         /// </returns>
415         [MethodImpl(MethodImplOptions.AggressiveInlining)]
416         protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
417     }
418 }
```

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

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

```
{
    /// <summary>
    /// <para>
    /// Represents the int 64 links size balanced tree methods base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksSizeBalancedTreeMethodsBase{ulong}"/>
    public unsafe abstract class UInt64LinksSizeBalancedTreeMethodsBase :
    ↪ LinksSizeBalancedTreeMethodsBase<ulong>
    {
        /// <summary>
        /// <para>
        /// The links.
        /// </para>
        /// <para></para>
        /// </summary>
        protected new readonly RawLink<ulong>* Links;
        /// <summary>
        /// <para>
        /// The header.
        /// </para>
        /// <para></para>
        /// </summary>
        protected new readonly LinksHeader<ulong>* Header;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt64LinksSizeBalancedTreeMethodsBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        protected UInt64LinksSizeBalancedTreeMethodsBase(LinksConstants<ulong> constants,
        ↪ RawLink<ulong>* links, LinksHeader<ulong>* header)
            : base(constants, (byte*)links, (byte*)header)
        {
            Links = links;
            Header = header;
        }

        /// <summary>
        /// <para>
        /// Gets the zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetZero() => 0UL;

        /// <summary>
        /// <para>
        /// Determines whether this instance equal to zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
```

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

        /// <summary>
        /// <para>
        /// Determines whether this instance are equal.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool AreEqual(ulong first, ulong second) => first == second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThanZero(ulong value) => value > 0UL;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThan(ulong first, ulong second) => first > second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater or equal than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
```

```
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater or equal than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
        ↪  always true for ulong

        /// <summary>
        /// <para>
        /// Determines whether this instance less or equal than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
        ↪  always >= 0 for ulong

        /// <summary>
        /// <para>
        /// Determines whether this instance less or equal than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;

        /// <summary>
        /// <para>
        /// Determines whether this instance less than zero.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
237         protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
            ↪  for ulong
238
239         /// <summary>
240         /// <para>
241         /// Determines whether this instance less than.
242         /// </para>
243         /// <para></para>
244         /// </summary>
245         /// <param name="first">
246         /// <para>The first.</para>
247         /// <para></para>
248         /// </param>
249         /// <param name="second">
250         /// <para>The second.</para>
251         /// <para></para>
252         /// </param>
253         /// <returns>
254         /// <para>The bool</para>
255         /// <para></para>
256         /// </returns>
257         [MethodImpl(MethodImplOptions.AggressiveInlining)]
258         protected override bool LessThan(ulong first, ulong second) => first < second;
259
260         /// <summary>
261         /// <para>
262         /// Increments the value.
263         /// </para>
264         /// <para></para>
265         /// </summary>
266         /// <param name="value">
267         /// <para>The value.</para>
268         /// <para></para>
269         /// </param>
270         /// <returns>
271         /// <para>The ulong</para>
272         /// <para></para>
273         /// </returns>
274         [MethodImpl(MethodImplOptions.AggressiveInlining)]
275         protected override ulong Increment(ulong value) => ++value;
276
277         /// <summary>
278         /// <para>
279         /// Decrements the value.
280         /// </para>
281         /// <para></para>
282         /// </summary>
283         /// <param name="value">
284         /// <para>The value.</para>
285         /// <para></para>
286         /// </param>
287         /// <returns>
288         /// <para>The ulong</para>
289         /// <para></para>
290         /// </returns>
291         [MethodImpl(MethodImplOptions.AggressiveInlining)]
292         protected override ulong Decrement(ulong value) => --value;
293
294         /// <summary>
295         /// <para>
296         /// Adds the first.
297         /// </para>
298         /// <para></para>
299         /// </summary>
300         /// <param name="first">
301         /// <para>The first.</para>
302         /// <para></para>
303         /// </param>
304         /// <param name="second">
305         /// <para>The second.</para>
306         /// <para></para>
307         /// </param>
308         /// <returns>
309         /// <para>The ulong</para>
310         /// <para></para>
311         /// </returns>
312         [MethodImpl(MethodImplOptions.AggressiveInlining)]
313         protected override ulong Add(ulong first, ulong second) => first + second;
```

```
314
315         /// <summary>
316         /// <para>
317         /// Subtracts the first.
318         /// </para>
319         /// <para></para>
320         /// </summary>
321         /// <param name="first">
322         /// <para>The first.</para>
323         /// <para></para>
324         /// </param>
325         /// <param name="second">
326         /// <para>The second.</para>
327         /// <para></para>
328         /// </param>
329         /// <returns>
330         /// <para>The ulong</para>
331         /// <para></para>
332         /// </returns>
333         [MethodImpl(MethodImplOptions.AggressiveInlining)]
334         protected override ulong Subtract(ulong first, ulong second) => first - second;
335
336         /// <summary>
337         /// <para>
338         /// Determines whether this instance first is to the left of second.
339         /// </para>
340         /// <para></para>
341         /// </summary>
342         /// <param name="first">
343         /// <para>The first.</para>
344         /// <para></para>
345         /// </param>
346         /// <param name="second">
347         /// <para>The second.</para>
348         /// <para></para>
349         /// </param>
350         /// <returns>
351         /// <para>The bool</para>
352         /// <para></para>
353         /// </returns>
354         [MethodImpl(MethodImplOptions.AggressiveInlining)]
355         protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
356         {
357             ref var firstLink = ref Links[first];
358             ref var secondLink = ref Links[second];
359             return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
                 ↪  secondLink.Source, secondLink.Target);
360         }
361
362         /// <summary>
363         /// <para>
364         /// Determines whether this instance first is to the right of second.
365         /// </para>
366         /// <para></para>
367         /// </summary>
368         /// <param name="first">
369         /// <para>The first.</para>
370         /// <para></para>
371         /// </param>
372         /// <param name="second">
373         /// <para>The second.</para>
374         /// <para></para>
375         /// </param>
376         /// <returns>
377         /// <para>The bool</para>
378         /// <para></para>
379         /// </returns>
380         [MethodImpl(MethodImplOptions.AggressiveInlining)]
381         protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
382         {
383             ref var firstLink = ref Links[first];
384             ref var secondLink = ref Links[second];
385             return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
                 ↪  secondLink.Source, secondLink.Target);
386         }
387
388         /// <summary>
389         /// <para>
```

```
390         /// Gets the header reference.
391         /// </para>
392         /// <para></para>
393         /// </summary>
394         /// <returns>
395         /// <para>A ref links header of ulong</para>
396         /// <para></para>
397         /// </returns>
398         [MethodImpl(MethodImplOptions.AggressiveInlining)]
399         protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
400
401         /// <summary>
402         /// <para>
403         /// Gets the link reference using the specified link.
404         /// </para>
405         /// <para></para>
406         /// </summary>
407         /// <param name="link">
408         /// <para>The link.</para>
409         /// <para></para>
410         /// </param>
411         /// <returns>
412         /// <para>A ref raw link of ulong</para>
413         /// <para></para>
414         /// </returns>
415         [MethodImpl(MethodImplOptions.AggressiveInlining)]
416         protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
417     }
418 }
```

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

```
1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Memory.United.Specific
6  {
7      /// <summary>
8      /// <para>
9      /// Represents the int 64 links sources avl balanced tree methods.
10     /// </para>
11     /// <para></para>
12     /// </summary>
13     /// <seealso cref="UInt64LinksAvlBalancedTreeMethodsBase"/>
14     public unsafe class UInt64LinksSourcesAvlBalancedTreeMethods :
       ↪  UInt64LinksAvlBalancedTreeMethodsBase
15     {
16         /// <summary>
17         /// <para>
18         /// Initializes a new <see cref="UInt64LinksSourcesAvlBalancedTreeMethods"/> instance.
19         /// </para>
20         /// <para></para>
21         /// </summary>
22         /// <param name="constants">
23         /// <para>A constants.</para>
24         /// <para></para>
25         /// </param>
26         /// <param name="links">
27         /// <para>A links.</para>
28         /// <para></para>
29         /// </param>
30         /// <param name="header">
31         /// <para>A header.</para>
32         /// <para></para>
33         /// </param>
34         public UInt64LinksSourcesAvlBalancedTreeMethods(LinksConstants<ulong> constants,
           ↪  RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
           ↪  { }
35
36         /// <summary>
37         /// <para>
38         /// Gets the left reference using the specified node.
39         /// </para>
40         /// <para></para>
41         /// </summary>
42         /// <param name="node">
43         /// <para>The node.</para>
44         /// <para></para>
```

```
45          /// </param>
46          /// <returns>
47          /// <para>The ref ulong</para>
48          /// <para></para>
49          /// </returns>
50          [MethodImpl(MethodImplOptions.AggressiveInlining)]
51          protected override ref ulong GetLeftReference(ulong node) => ref
            ↪ Links[node].LeftAsSource;
52
53          /// <summary>
54          /// <para>
55          /// Gets the right reference using the specified node.
56          /// </para>
57          /// <para></para>
58          /// </summary>
59          /// <param name="node">
60          /// <para>The node.</para>
61          /// <para></para>
62          /// </param>
63          /// <returns>
64          /// <para>The ref ulong</para>
65          /// <para></para>
66          /// </returns>
67          [MethodImpl(MethodImplOptions.AggressiveInlining)]
68          protected override ref ulong GetRightReference(ulong node) => ref
            ↪ Links[node].RightAsSource;
69
70          /// <summary>
71          /// <para>
72          /// Gets the left using the specified node.
73          /// </para>
74          /// <para></para>
75          /// </summary>
76          /// <param name="node">
77          /// <para>The node.</para>
78          /// <para></para>
79          /// </param>
80          /// <returns>
81          /// <para>The ulong</para>
82          /// <para></para>
83          /// </returns>
84          [MethodImpl(MethodImplOptions.AggressiveInlining)]
85          protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;
86
87          /// <summary>
88          /// <para>
89          /// Gets the right using the specified node.
90          /// </para>
91          /// <para></para>
92          /// </summary>
93          /// <param name="node">
94          /// <para>The node.</para>
95          /// <para></para>
96          /// </param>
97          /// <returns>
98          /// <para>The ulong</para>
99          /// <para></para>
100         /// </returns>
101         [MethodImpl(MethodImplOptions.AggressiveInlining)]
102         protected override ulong GetRight(ulong node) => Links[node].RightAsSource;
103
104         /// <summary>
105         /// <para>
106         /// Sets the left using the specified node.
107         /// </para>
108         /// <para></para>
109         /// </summary>
110         /// <param name="node">
111         /// <para>The node.</para>
112         /// <para></para>
113         /// </param>
114         /// <param name="left">
115         /// <para>The left.</para>
116         /// <para></para>
117         /// </param>
118         [MethodImpl(MethodImplOptions.AggressiveInlining)]
119         protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
            ↪ left;
```

```
120
121         /// <summary>
122         /// <para>
123         /// Sets the right using the specified node.
124         /// </para>
125         /// <para></para>
126         /// </summary>
127         /// <param name="node">
128         /// <para>The node.</para>
129         /// <para></para>
130         /// </param>
131         /// <param name="right">
132         /// <para>The right.</para>
133         /// <para></para>
134         /// </param>
135         [MethodImpl(MethodImplOptions.AggressiveInlining)]
136         protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
            ↪ right;
137
138         /// <summary>
139         /// <para>
140         /// Gets the size using the specified node.
141         /// </para>
142         /// <para></para>
143         /// </summary>
144         /// <param name="node">
145         /// <para>The node.</para>
146         /// <para></para>
147         /// </param>
148         /// <returns>
149         /// <para>The ulong</para>
150         /// <para></para>
151         /// </returns>
152         [MethodImpl(MethodImplOptions.AggressiveInlining)]
153         protected override ulong GetSize(ulong node) => GetSizeValue(Links[node].SizeAsSource);
154
155         /// <summary>
156         /// <para>
157         /// Sets the size using the specified node.
158         /// </para>
159         /// <para></para>
160         /// </summary>
161         /// <param name="node">
162         /// <para>The node.</para>
163         /// <para></para>
164         /// </param>
165         /// <param name="size">
166         /// <para>The size.</para>
167         /// <para></para>
168         /// </param>
169         [MethodImpl(MethodImplOptions.AggressiveInlining)]
170         protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
            ↪ Links[node].SizeAsSource, size);
171
172         /// <summary>
173         /// <para>
174         /// Determines whether this instance get left is child.
175         /// </para>
176         /// <para></para>
177         /// </summary>
178         /// <param name="node">
179         /// <para>The node.</para>
180         /// <para></para>
181         /// </param>
182         /// <returns>
183         /// <para>The bool</para>
184         /// <para></para>
185         /// </returns>
186         [MethodImpl(MethodImplOptions.AggressiveInlining)]
187         protected override bool GetLeftIsChild(ulong node) =>
            ↪ GetLeftIsChildValue(Links[node].SizeAsSource);
188
189         //[MethodImpl(MethodImplOptions.AggressiveInlining)]
190         //protected override bool GetLeftIsChild(ulong node) => IsChild(node, GetLeft(node));
191
192         /// <summary>
193         /// <para>
194         /// Sets the left is child using the specified node.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeftIsChild(ulong node, bool value) =>
        ↪ SetLeftIsChildValue(ref Links[node].SizeAsSource, value);

        /// <summary>
        /// <para>
        /// Determines whether this instance get right is child.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [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));

        /// <summary>
        /// <para>
        /// Sets the right is child using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRightIsChild(ulong node, bool value) =>
        ↪ SetRightIsChildValue(ref Links[node].SizeAsSource, value);

        /// <summary>
        /// <para>
        /// Gets the balance using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The sbyte</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override sbyte GetBalance(ulong node) =>
        ↪ GetBalanceValue(Links[node].SizeAsSource);

        /// <summary>
        /// <para>
        /// Sets the balance using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
```

```
269         /// <param name="node">
270         /// <para>The node.</para>
271         /// <para></para>
272         /// </param>
273         /// <param name="value">
274         /// <para>The value.</para>
275         /// <para></para>
276         /// </param>
277         [MethodImpl(MethodImplOptions.AggressiveInlining)]
278         protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
            ↪ Links[node].SizeAsSource, value);
279
280         /// <summary>
281         /// <para>
282         /// Gets the tree root.
283         /// </para>
284         /// <para></para>
285         /// </summary>
286         /// <returns>
287         /// <para>The ulong</para>
288         /// <para></para>
289         /// </returns>
290         [MethodImpl(MethodImplOptions.AggressiveInlining)]
291         protected override ulong GetTreeRoot() => Header->RootAsSource;
292
293         /// <summary>
294         /// <para>
295         /// Gets the base part value using the specified link.
296         /// </para>
297         /// <para></para>
298         /// </summary>
299         /// <param name="link">
300         /// <para>The link.</para>
301         /// <para></para>
302         /// </param>
303         /// <returns>
304         /// <para>The ulong</para>
305         /// <para></para>
306         /// </returns>
307         [MethodImpl(MethodImplOptions.AggressiveInlining)]
308         protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
309
310         /// <summary>
311         /// <para>
312         /// Determines whether this instance first is to the left of second.
313         /// </para>
314         /// <para></para>
315         /// </summary>
316         /// <param name="firstSource">
317         /// <para>The first source.</para>
318         /// <para></para>
319         /// </param>
320         /// <param name="firstTarget">
321         /// <para>The first target.</para>
322         /// <para></para>
323         /// </param>
324         /// <param name="secondSource">
325         /// <para>The second source.</para>
326         /// <para></para>
327         /// </param>
328         /// <param name="secondTarget">
329         /// <para>The second target.</para>
330         /// <para></para>
331         /// </param>
332         /// <returns>
333         /// <para>The bool</para>
334         /// <para></para>
335         /// </returns>
336         [MethodImpl(MethodImplOptions.AggressiveInlining)]
337         protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
            ↪ ulong secondSource, ulong secondTarget)
338             => firstSource < secondSource || (firstSource == secondSource && firstTarget <
                ↪ secondTarget);
339
340         /// <summary>
341         /// <para>
342         /// Determines whether this instance first is to the right of second.
343         /// </para>
```

```
344         /// <para></para>
345         /// </summary>
346         /// <param name="firstSource">
347         /// <para>The first source.</para>
348         /// <para></para>
349         /// </param>
350         /// <param name="firstTarget">
351         /// <para>The first target.</para>
352         /// <para></para>
353         /// </param>
354         /// <param name="secondSource">
355         /// <para>The second source.</para>
356         /// <para></para>
357         /// </param>
358         /// <param name="secondTarget">
359         /// <para>The second target.</para>
360         /// <para></para>
361         /// </param>
362         /// <returns>
363         /// <para>The bool</para>
364         /// <para></para>
365         /// </returns>
366         [MethodImpl(MethodImplOptions.AggressiveInlining)]
367         protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
          ↪  ulong secondSource, ulong secondTarget)
368             => firstSource > secondSource || (firstSource == secondSource && firstTarget >
              ↪  secondTarget);
369
370         /// <summary>
371         /// <para>
372         /// Clears the node using the specified node.
373         /// </para>
374         /// <para></para>
375         /// </summary>
376         /// <param name="node">
377         /// <para>The node.</para>
378         /// <para></para>
379         /// </param>
380         [MethodImpl(MethodImplOptions.AggressiveInlining)]
381         protected override void ClearNode(ulong node)
382         {
383             ref var link = ref Links[node];
384             link.LeftAsSource = 0UL;
385             link.RightAsSource = 0UL;
386             link.SizeAsSource = 0UL;
387         }
388     }
389 }
```

1.105 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesRecursionlessSizeBalancedTr

```
1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Memory.United.Specific
6  {
7      /// <summary>
8      /// <para>
9      /// Represents the int 64 links sources recursionless size balanced tree methods.
10     /// </para>
11     /// <para></para>
12     /// </summary>
13     /// <seealso cref="UInt64LinksRecursionlessSizeBalancedTreeMethodsBase"/>
14     public unsafe class UInt64LinksSourcesRecursionlessSizeBalancedTreeMethods :
       ↪  UInt64LinksRecursionlessSizeBalancedTreeMethodsBase
15     {
16         /// <summary>
17         /// <para>
18         /// Initializes a new <see
           ↪  cref="UInt64LinksSourcesRecursionlessSizeBalancedTreeMethods"/> instance.
19         /// </para>
20         /// <para></para>
21         /// </summary>
22         /// <param name="constants">
23         /// <para>A constants.</para>
24         /// <para></para>
25         /// </param>
26         /// <param name="links">
```

```
27          /// <para>A links.</para>
28          /// <para></para>
29          /// </param>
30          /// <param name="header">
31          /// <para>A header.</para>
32          /// <para></para>
33          /// </param>
34          public UInt64LinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<ulong>
            ↪  constants, RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants,
            ↪  links, header) { }
35
36          /// <summary>
37          /// <para>
38          /// Gets the left reference using the specified node.
39          /// </para>
40          /// <para></para>
41          /// </summary>
42          /// <param name="node">
43          /// <para>The node.</para>
44          /// <para></para>
45          /// </param>
46          /// <returns>
47          /// <para>The ref ulong</para>
48          /// <para></para>
49          /// </returns>
50          [MethodImpl(MethodImplOptions.AggressiveInlining)]
51          protected override ref ulong GetLeftReference(ulong node) => ref
            ↪  Links[node].LeftAsSource;
52
53          /// <summary>
54          /// <para>
55          /// Gets the right reference using the specified node.
56          /// </para>
57          /// <para></para>
58          /// </summary>
59          /// <param name="node">
60          /// <para>The node.</para>
61          /// <para></para>
62          /// </param>
63          /// <returns>
64          /// <para>The ref ulong</para>
65          /// <para></para>
66          /// </returns>
67          [MethodImpl(MethodImplOptions.AggressiveInlining)]
68          protected override ref ulong GetRightReference(ulong node) => ref
            ↪  Links[node].RightAsSource;
69
70          /// <summary>
71          /// <para>
72          /// Gets the left using the specified node.
73          /// </para>
74          /// <para></para>
75          /// </summary>
76          /// <param name="node">
77          /// <para>The node.</para>
78          /// <para></para>
79          /// </param>
80          /// <returns>
81          /// <para>The ulong</para>
82          /// <para></para>
83          /// </returns>
84          [MethodImpl(MethodImplOptions.AggressiveInlining)]
85          protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;
86
87          /// <summary>
88          /// <para>
89          /// Gets the right using the specified node.
90          /// </para>
91          /// <para></para>
92          /// </summary>
93          /// <param name="node">
94          /// <para>The node.</para>
95          /// <para></para>
96          /// </param>
97          /// <returns>
98          /// <para>The ulong</para>
99          /// <para></para>
100         /// </returns>
```

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

        /// <summary>
        /// <para>
        /// Sets the left using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="left">
        /// <para>The left.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
        ↪  left;

        /// <summary>
        /// <para>
        /// Sets the right using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="right">
        /// <para>The right.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
        ↪  right;

        /// <summary>
        /// <para>
        /// Gets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetSize(ulong node) => Links[node].SizeAsSource;

        /// <summary>
        /// <para>
        /// Sets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="size">
        /// <para>The size.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsSource =
        ↪  size;

        /// <summary>
        /// <para>
        /// Gets the tree root.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetTreeRoot() => Header->RootAsSource;

        /// <summary>
        /// <para>
        /// Gets the base part value using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetBasePartValue(ulong link) => Links[link].Source;

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="firstSource">
        /// <para>The first source.</para>
        /// <para></para>
        /// </param>
        /// <param name="firstTarget">
        /// <para>The first target.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondSource">
        /// <para>The second source.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondTarget">
        /// <para>The second target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
        ↪  ulong secondSource, ulong secondTarget)
            => firstSource < secondSource || (firstSource == secondSource && firstTarget <
            ↪  secondTarget);

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the right of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="firstSource">
        /// <para>The first source.</para>
        /// <para></para>
        /// </param>
        /// <param name="firstTarget">
        /// <para>The first target.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondSource">
        /// <para>The second source.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondTarget">
        /// <para>The second target.</para>
```

```
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
        ↪  ulong secondSource, ulong secondTarget)
            => firstSource > secondSource || (firstSource == secondSource && firstTarget >
            ↪  secondTarget);

        /// <summary>
        /// <para>
        /// Clears the node using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(ulong node)
        {
            ref var link = ref Links[node];
            link.LeftAsSource = OUL;
            link.RightAsSource = OUL;
            link.SizeAsSource = OUL;
        }
    }
}
```

1.106 ./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
{
    /// <summary>
    /// <para>
    /// Represents the int 64 links sources size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="UInt64LinksSizeBalancedTreeMethodsBase"/>
    public unsafe class UInt64LinksSourcesSizeBalancedTreeMethods :
    ↪  UInt64LinksSizeBalancedTreeMethodsBase
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt64LinksSourcesSizeBalancedTreeMethods"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        public UInt64LinksSourcesSizeBalancedTreeMethods(LinksConstants<ulong> constants,
        ↪  RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
        ↪  { }

        /// <summary>
        /// <para>
        /// Gets the left reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
```

```
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetLeftReference(ulong node) => ref
        ↪ Links[node].LeftAsSource;

        /// <summary>
        /// <para>
        /// Gets the right reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetRightReference(ulong node) => ref
        ↪ Links[node].RightAsSource;

        /// <summary>
        /// <para>
        /// Gets the left using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;

        /// <summary>
        /// <para>
        /// Gets the right using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetRight(ulong node) => Links[node].RightAsSource;

        /// <summary>
        /// <para>
        /// Sets the left using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="left">
        /// <para>The left.</para>
        /// <para></para>
        /// </param>
```

```
118             [MethodImpl(MethodImplOptions.AggressiveInlining)]
119             protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
                ↪ left;
120
121             /// <summary>
122             /// <para>
123             /// Sets the right using the specified node.
124             /// </para>
125             /// <para></para>
126             /// </summary>
127             /// <param name="node">
128             /// <para>The node.</para>
129             /// <para></para>
130             /// </param>
131             /// <param name="right">
132             /// <para>The right.</para>
133             /// <para></para>
134             /// </param>
135             [MethodImpl(MethodImplOptions.AggressiveInlining)]
136             protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
                ↪ right;
137
138             /// <summary>
139             /// <para>
140             /// Gets the size using the specified node.
141             /// </para>
142             /// <para></para>
143             /// </summary>
144             /// <param name="node">
145             /// <para>The node.</para>
146             /// <para></para>
147             /// </param>
148             /// <returns>
149             /// <para>The ulong</para>
150             /// <para></para>
151             /// </returns>
152             [MethodImpl(MethodImplOptions.AggressiveInlining)]
153             protected override ulong GetSize(ulong node) => Links[node].SizeAsSource;
154
155             /// <summary>
156             /// <para>
157             /// Sets the size using the specified node.
158             /// </para>
159             /// <para></para>
160             /// </summary>
161             /// <param name="node">
162             /// <para>The node.</para>
163             /// <para></para>
164             /// </param>
165             /// <param name="size">
166             /// <para>The size.</para>
167             /// <para></para>
168             /// </param>
169             [MethodImpl(MethodImplOptions.AggressiveInlining)]
170             protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsSource =
                ↪ size;
171
172             /// <summary>
173             /// <para>
174             /// Gets the tree root.
175             /// </para>
176             /// <para></para>
177             /// </summary>
178             /// <returns>
179             /// <para>The ulong</para>
180             /// <para></para>
181             /// </returns>
182             [MethodImpl(MethodImplOptions.AggressiveInlining)]
183             protected override ulong GetTreeRoot() => Header->RootAsSource;
184
185             /// <summary>
186             /// <para>
187             /// Gets the base part value using the specified link.
188             /// </para>
189             /// <para></para>
190             /// </summary>
191             /// <param name="link">
192             /// <para>The link.</para>
```

```
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetBasePartValue(ulong link) => Links[link].Source;

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="firstSource">
        /// <para>The first source.</para>
        /// <para></para>
        /// </param>
        /// <param name="firstTarget">
        /// <para>The first target.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondSource">
        /// <para>The second source.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondTarget">
        /// <para>The second target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
        ↪  ulong secondSource, ulong secondTarget)
            => firstSource < secondSource || (firstSource == secondSource && firstTarget <
            ↪  secondTarget);

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the right of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="firstSource">
        /// <para>The first source.</para>
        /// <para></para>
        /// </param>
        /// <param name="firstTarget">
        /// <para>The first target.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondSource">
        /// <para>The second source.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondTarget">
        /// <para>The second target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
        ↪  ulong secondSource, ulong secondTarget)
            => firstSource > secondSource || (firstSource == secondSource && firstTarget >
            ↪  secondTarget);

        /// <summary>
        /// <para>
        /// Clears the node using the specified node.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(ulong node)
        {
            ref var link = ref Links[node];
            link.LeftAsSource = OUL;
            link.RightAsSource = OUL;
            link.SizeAsSource = OUL;
        }
    }
}
```

1.107 ./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
{
    /// <summary>
    /// <para>
    /// Represents the int 64 links targets avl balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="UInt64LinksAvlBalancedTreeMethodsBase"/>
    public unsafe class UInt64LinksTargetsAvlBalancedTreeMethods :
    ↪ UInt64LinksAvlBalancedTreeMethodsBase
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt64LinksTargetsAvlBalancedTreeMethods"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        public UInt64LinksTargetsAvlBalancedTreeMethods(LinksConstants<ulong> constants,
        ↪ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
        ↪ { }

        /// <summary>
        /// <para>
        /// Gets the left reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetLeftReference(ulong node) => ref
        ↪ Links[node].LeftAsTarget;

        /// <summary>
        /// <para>
        /// Gets the right reference using the specified node.
        /// </para>
```

```
57          /// <para></para>
58          /// </summary>
59          /// <param name="node">
60          /// <para>The node.</para>
61          /// <para></para>
62          /// </param>
63          /// <returns>
64          /// <para>The ref ulong</para>
65          /// <para></para>
66          /// </returns>
67          [MethodImpl(MethodImplOptions.AggressiveInlining)]
68          protected override ref ulong GetRightReference(ulong node) => ref
            ↪ Links[node].RightAsTarget;
69
70          /// <summary>
71          /// <para>
72          /// Gets the left using the specified node.
73          /// </para>
74          /// <para></para>
75          /// </summary>
76          /// <param name="node">
77          /// <para>The node.</para>
78          /// <para></para>
79          /// </param>
80          /// <returns>
81          /// <para>The ulong</para>
82          /// <para></para>
83          /// </returns>
84          [MethodImpl(MethodImplOptions.AggressiveInlining)]
85          protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;
86
87          /// <summary>
88          /// <para>
89          /// Gets the right using the specified node.
90          /// </para>
91          /// <para></para>
92          /// </summary>
93          /// <param name="node">
94          /// <para>The node.</para>
95          /// <para></para>
96          /// </param>
97          /// <returns>
98          /// <para>The ulong</para>
99          /// <para></para>
100         /// </returns>
101         [MethodImpl(MethodImplOptions.AggressiveInlining)]
102         protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;
103
104         /// <summary>
105         /// <para>
106         /// Sets the left using the specified node.
107         /// </para>
108         /// <para></para>
109         /// </summary>
110         /// <param name="node">
111         /// <para>The node.</para>
112         /// <para></para>
113         /// </param>
114         /// <param name="left">
115         /// <para>The left.</para>
116         /// <para></para>
117         /// </param>
118         [MethodImpl(MethodImplOptions.AggressiveInlining)]
119         protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
            ↪ left;
120
121         /// <summary>
122         /// <para>
123         /// Sets the right using the specified node.
124         /// </para>
125         /// <para></para>
126         /// </summary>
127         /// <param name="node">
128         /// <para>The node.</para>
129         /// <para></para>
130         /// </param>
131         /// <param name="right">
132         /// <para>The right.</para>
```

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

        /// <summary>
        /// <para>
        /// Gets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetSize(ulong node) => GetSizeValue(Links[node].SizeAsTarget);

        /// <summary>
        /// <para>
        /// Sets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="size">
        /// <para>The size.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
        ↪  Links[node].SizeAsTarget, size);

        /// <summary>
        /// <para>
        /// Determines whether this instance get left is child.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetLeftIsChild(ulong node) =>
        ↪  GetLeftIsChildValue(Links[node].SizeAsTarget);

        /// <summary>
        /// <para>
        /// Sets the left is child using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeftIsChild(ulong node, bool value) =>
        ↪  SetLeftIsChildValue(ref Links[node].SizeAsTarget, value);

        /// <summary>
```

```
        /// <para>
        /// Determines whether this instance get right is child.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetRightIsChild(ulong node) =>
        ↪  GetRightIsChildValue(Links[node].SizeAsTarget);

        /// <summary>
        /// <para>
        /// Sets the right is child using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRightIsChild(ulong node, bool value) =>
        ↪  SetRightIsChildValue(ref Links[node].SizeAsTarget, value);

        /// <summary>
        /// <para>
        /// Gets the balance using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The sbyte</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override sbyte GetBalance(ulong node) =>
        ↪  GetBalanceValue(Links[node].SizeAsTarget);

        /// <summary>
        /// <para>
        /// Sets the balance using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
        ↪  Links[node].SizeAsTarget, value);

        /// <summary>
        /// <para>
        /// Gets the tree root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
```

```
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetTreeRoot() => Header->RootAsTarget;

        /// <summary>
        /// <para>
        /// Gets the base part value using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetBasePartValue(ulong link) => Links[link].Target;

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="firstSource">
        /// <para>The first source.</para>
        /// <para></para>
        /// </param>
        /// <param name="firstTarget">
        /// <para>The first target.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondSource">
        /// <para>The second source.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondTarget">
        /// <para>The second target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
        ↪  ulong secondSource, ulong secondTarget)
            => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
            ↪  secondSource);

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the right of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="firstSource">
        /// <para>The first source.</para>
        /// <para></para>
        /// </param>
        /// <param name="firstTarget">
        /// <para>The first target.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondSource">
        /// <para>The second source.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondTarget">
        /// <para>The second target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
```

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

        /// <summary>
        /// <para>
        /// Clears the node using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(ulong node)
        {
            ref var link = ref Links[node];
            link.LeftAsTarget = OUL;
            link.RightAsTarget = OUL;
            link.SizeAsTarget = OUL;
        }
    }
}
```

1.108 ./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
{
    /// <summary>
    /// <para>
    /// Represents the int 64 links targets recursionless size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="UInt64LinksRecursionlessSizeBalancedTreeMethodsBase"/>
    public unsafe class UInt64LinksTargetsRecursionlessSizeBalancedTreeMethods :
    ↪  UInt64LinksRecursionlessSizeBalancedTreeMethodsBase
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see
        ↪  cref="UInt64LinksTargetsRecursionlessSizeBalancedTreeMethods"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        public UInt64LinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<ulong>
        ↪  constants, RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants,
        ↪  links, header) { }

        /// <summary>
        /// <para>
        /// Gets the left reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
```

```
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetLeftReference(ulong node) => ref
        ↪  Links[node].LeftAsTarget;

        /// <summary>
        /// <para>
        /// Gets the right reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetRightReference(ulong node) => ref
        ↪  Links[node].RightAsTarget;

        /// <summary>
        /// <para>
        /// Gets the left using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;

        /// <summary>
        /// <para>
        /// Gets the right using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;

        /// <summary>
        /// <para>
        /// Sets the left using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="left">
        /// <para>The left.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

        /// <summary>
        /// <para>
        /// Sets the right using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="right">
        /// <para>The right.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
        ↪ right;

        /// <summary>
        /// <para>
        /// Gets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;

        /// <summary>
        /// <para>
        /// Sets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="size">
        /// <para>The size.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsTarget =
        ↪ size;

        /// <summary>
        /// <para>
        /// Gets the tree root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetTreeRoot() => Header->RootAsTarget;

        /// <summary>
        /// <para>
        /// Gets the base part value using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
```

```
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetBasePartValue(ulong link) => Links[link].Target;

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="firstSource">
        /// <para>The first source.</para>
        /// <para></para>
        /// </param>
        /// <param name="firstTarget">
        /// <para>The first target.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondSource">
        /// <para>The second source.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondTarget">
        /// <para>The second target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
        ↪  ulong secondSource, ulong secondTarget)
            => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
            ↪  secondSource);

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the right of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="firstSource">
        /// <para>The first source.</para>
        /// <para></para>
        /// </param>
        /// <param name="firstTarget">
        /// <para>The first target.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondSource">
        /// <para>The second source.</para>
        /// <para></para>
        /// </param>
        /// <param name="secondTarget">
        /// <para>The second target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
        ↪  ulong secondSource, ulong secondTarget)
            => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
            ↪  secondSource);

        /// <summary>
        /// <para>
        /// Clears the node using the specified node.
        /// </para>
        /// <para></para>
```

```
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(ulong node)
        {
            ref var link = ref Links[node];
            link.LeftAsTarget = OUL;
            link.RightAsTarget = OUL;
            link.SizeAsTarget = OUL;
        }
    }
}
```

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

```
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Memory.United.Specific
{
    /// <summary>
    /// <para>
    /// Represents the int 64 links targets size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="UInt64LinksSizeBalancedTreeMethodsBase"/>
    public unsafe class UInt64LinksTargetsSizeBalancedTreeMethods :
    ↪ UInt64LinksSizeBalancedTreeMethodsBase
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt64LinksTargetsSizeBalancedTreeMethods"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        public UInt64LinksTargetsSizeBalancedTreeMethods(LinksConstants<ulong> constants,
        ↪ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
        ↪ { }

        /// <summary>
        /// <para>
        /// Gets the left reference using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ref ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetLeftReference(ulong node) => ref
        ↪ Links[node].LeftAsTarget;

        /// <summary>
        /// <para>
        /// Gets the right reference using the specified node.
        /// </para>
        /// <para></para>
```

```
58          /// </summary>
59          /// <param name="node">
60          /// <para>The node.</para>
61          /// <para></para>
62          /// </param>
63          /// <returns>
64          /// <para>The ref ulong</para>
65          /// <para></para>
66          /// </returns>
67          [MethodImpl(MethodImplOptions.AggressiveInlining)]
68          protected override ref ulong GetRightReference(ulong node) => ref
            ↪  Links[node].RightAsTarget;
69
70          /// <summary>
71          /// <para>
72          /// Gets the left using the specified node.
73          /// </para>
74          /// <para></para>
75          /// </summary>
76          /// <param name="node">
77          /// <para>The node.</para>
78          /// <para></para>
79          /// </param>
80          /// <returns>
81          /// <para>The ulong</para>
82          /// <para></para>
83          /// </returns>
84          [MethodImpl(MethodImplOptions.AggressiveInlining)]
85          protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;
86
87          /// <summary>
88          /// <para>
89          /// Gets the right using the specified node.
90          /// </para>
91          /// <para></para>
92          /// </summary>
93          /// <param name="node">
94          /// <para>The node.</para>
95          /// <para></para>
96          /// </param>
97          /// <returns>
98          /// <para>The ulong</para>
99          /// <para></para>
100         /// </returns>
101         [MethodImpl(MethodImplOptions.AggressiveInlining)]
102         protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;
103
104         /// <summary>
105         /// <para>
106         /// Sets the left using the specified node.
107         /// </para>
108         /// <para></para>
109         /// </summary>
110         /// <param name="node">
111         /// <para>The node.</para>
112         /// <para></para>
113         /// </param>
114         /// <param name="left">
115         /// <para>The left.</para>
116         /// <para></para>
117         /// </param>
118         [MethodImpl(MethodImplOptions.AggressiveInlining)]
119         protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
            ↪  left;
120
121         /// <summary>
122         /// <para>
123         /// Sets the right using the specified node.
124         /// </para>
125         /// <para></para>
126         /// </summary>
127         /// <param name="node">
128         /// <para>The node.</para>
129         /// <para></para>
130         /// </param>
131         /// <param name="right">
132         /// <para>The right.</para>
133         /// <para></para>
```

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

        /// <summary>
        /// <para>
        /// Gets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;

        /// <summary>
        /// <para>
        /// Sets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <param name="size">
        /// <para>The size.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsTarget =
        ↪  size;

        /// <summary>
        /// <para>
        /// Gets the tree root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetTreeRoot() => Header->RootAsTarget;

        /// <summary>
        /// <para>
        /// Gets the base part value using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetBasePartValue(ulong link) => Links[link].Target;

        /// <summary>
        /// <para>
        /// Determines whether this instance first is to the left of second.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="firstSource">
        /// <para>The first source.</para>
```

```
210             /// <para></para>
211             /// </param>
212             /// <param name="firstTarget">
213             /// <para>The first target.</para>
214             /// <para></para>
215             /// </param>
216             /// <param name="secondSource">
217             /// <para>The second source.</para>
218             /// <para></para>
219             /// </param>
220             /// <param name="secondTarget">
221             /// <para>The second target.</para>
222             /// <para></para>
223             /// </param>
224             /// <returns>
225             /// <para>The bool</para>
226             /// <para></para>
227             /// </returns>
228             [MethodImpl(MethodImplOptions.AggressiveInlining)]
229             protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
                ↪  ulong secondSource, ulong secondTarget)
230                 => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
                    ↪  secondSource);
231
232             /// <summary>
233             /// <para>
234             /// Determines whether this instance first is to the right of second.
235             /// </para>
236             /// <para></para>
237             /// </summary>
238             /// <param name="firstSource">
239             /// <para>The first source.</para>
240             /// <para></para>
241             /// </param>
242             /// <param name="firstTarget">
243             /// <para>The first target.</para>
244             /// <para></para>
245             /// </param>
246             /// <param name="secondSource">
247             /// <para>The second source.</para>
248             /// <para></para>
249             /// </param>
250             /// <param name="secondTarget">
251             /// <para>The second target.</para>
252             /// <para></para>
253             /// </param>
254             /// <returns>
255             /// <para>The bool</para>
256             /// <para></para>
257             /// </returns>
258             [MethodImpl(MethodImplOptions.AggressiveInlining)]
259             protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
                ↪  ulong secondSource, ulong secondTarget)
260                 => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
                    ↪  secondSource);
261
262             /// <summary>
263             /// <para>
264             /// Clears the node using the specified node.
265             /// </para>
266             /// <para></para>
267             /// </summary>
268             /// <param name="node">
269             /// <para>The node.</para>
270             /// <para></para>
271             /// </param>
272             [MethodImpl(MethodImplOptions.AggressiveInlining)]
273             protected override void ClearNode(ulong node)
274             {
275                 ref var link = ref Links[node];
276                 link.LeftAsTarget = OUL;
277                 link.RightAsTarget = OUL;
278                 link.SizeAsTarget = OUL;
279             }
280         }
281     }
```

```
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;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt64UnitedMemoryLinks"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="address">
        /// <para>A address.</para>
        /// <para></para>
        /// </param>
        [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) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt64UnitedMemoryLinks"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="memory">
        /// <para>A memory.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory, DefaultLinksSizeStep) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt64UnitedMemoryLinks"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="memory">
        /// <para>A memory.</para>
        /// <para></para>
        /// </param>
        /// <param name="memoryReservationStep">
        /// <para>A memory reservation step.</para>
        /// <para></para>
```

```
69          /// </param>
70          [MethodImpl(MethodImplOptions.AggressiveInlining)]
71          public UInt64UnitedMemoryLinks(IResizableDirectMemory memory, long
            ↪  memoryReservationStep) : this(memory, memoryReservationStep,
            ↪  Default<LinksConstants<ulong>>.Instance, IndexTreeType.Default) { }
72
73          /// <summary>
74          /// <para>
75          /// Initializes a new <see cref="UInt64UnitedMemoryLinks"/> instance.
76          /// </para>
77          /// <para></para>
78          /// </summary>
79          /// <param name="memory">
80          /// <para>A memory.</para>
81          /// <para></para>
82          /// </param>
83          /// <param name="memoryReservationStep">
84          /// <para>A memory reservation step.</para>
85          /// <para></para>
86          /// </param>
87          /// <param name="constants">
88          /// <para>A constants.</para>
89          /// <para></para>
90          /// </param>
91          /// <param name="indexTreeType">
92          /// <para>A index tree type.</para>
93          /// <para></para>
94          /// </param>
95          [MethodImpl(MethodImplOptions.AggressiveInlining)]
96          public UInt64UnitedMemoryLinks(IResizableDirectMemory memory, long
            ↪  memoryReservationStep, LinksConstants<ulong> constants, IndexTreeType indexTreeType)
            ↪  : base(memory, memoryReservationStep, constants)
97          {
98              if (indexTreeType == IndexTreeType.SizedAndThreadedAVLBalancedTree)
99              {
100                 _createSourceTreeMethods = () => new
                    ↪  UInt64LinksSourcesAvlBalancedTreeMethods(Constants, _links, _header);
101                 _createTargetTreeMethods = () => new
                    ↪  UInt64LinksTargetsAvlBalancedTreeMethods(Constants, _links, _header);
102             }
103             else if (indexTreeType == IndexTreeType.SizeBalancedTree)
104             {
105                 _createSourceTreeMethods = () => new
                    ↪  UInt64LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
106                 _createTargetTreeMethods = () => new
                    ↪  UInt64LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
107             }
108             else
109             {
110                 _createSourceTreeMethods = () => new
                    ↪  UInt64LinksSourcesRecursionlessSizeBalancedTreeMethods(Constants, _links,
                    ↪  _header);
111                 _createTargetTreeMethods = () => new
                    ↪  UInt64LinksTargetsRecursionlessSizeBalancedTreeMethods(Constants, _links,
                    ↪  _header);
112             }
113             Init(memory, memoryReservationStep);
114         }
115
116         /// <summary>
117         /// <para>
118         /// Sets the pointers using the specified memory.
119         /// </para>
120         /// <para></para>
121         /// </summary>
122         /// <param name="memory">
123         /// <para>The memory.</para>
124         /// <para></para>
125         /// </param>
126         [MethodImpl(MethodImplOptions.AggressiveInlining)]
127         protected override void SetPointers(IResizableDirectMemory memory)
128         {
129             _header = (LinksHeader<ulong>*)memory.Pointer;
130             _links = (RawLink<ulong>*)memory.Pointer;
131             SourcesTreeMethods = _createSourceTreeMethods();
132             TargetsTreeMethods = _createTargetTreeMethods();
133             UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_links, _header);
```

```
134         }
135
136         /// <summary>
137         /// <para>
138         /// Resets the pointers.
139         /// </para>
140         /// <para></para>
141         /// </summary>
142         [MethodImpl(MethodImplOptions.AggressiveInlining)]
143         protected override void ResetPointers()
144         {
145             base.ResetPointers();
146             _links = null;
147             _header = null;
148         }
149
150         /// <summary>
151         /// <para>
152         /// Gets the header reference.
153         /// </para>
154         /// <para></para>
155         /// </summary>
156         /// <returns>
157         /// <para>A ref links header of ulong</para>
158         /// <para></para>
159         /// </returns>
160         [MethodImpl(MethodImplOptions.AggressiveInlining)]
161         protected override ref LinksHeader<ulong> GetHeaderReference() => ref *_header;
162
163         /// <summary>
164         /// <para>
165         /// Gets the link reference using the specified link index.
166         /// </para>
167         /// <para></para>
168         /// </summary>
169         /// <param name="linkIndex">
170         /// <para>The link index.</para>
171         /// <para></para>
172         /// </param>
173         /// <returns>
174         /// <para>A ref raw link of ulong</para>
175         /// <para></para>
176         /// </returns>
177         [MethodImpl(MethodImplOptions.AggressiveInlining)]
178         protected override ref RawLink<ulong> GetLinkReference(ulong linkIndex) => ref
            ↪  _links[linkIndex];
179
180         /// <summary>
181         /// <para>
182         /// Determines whether this instance are equal.
183         /// </para>
184         /// <para></para>
185         /// </summary>
186         /// <param name="first">
187         /// <para>The first.</para>
188         /// <para></para>
189         /// </param>
190         /// <param name="second">
191         /// <para>The second.</para>
192         /// <para></para>
193         /// </param>
194         /// <returns>
195         /// <para>The bool</para>
196         /// <para></para>
197         /// </returns>
198         [MethodImpl(MethodImplOptions.AggressiveInlining)]
199         protected override bool AreEqual(ulong first, ulong second) => first == second;
200
201         /// <summary>
202         /// <para>
203         /// Determines whether this instance less than.
204         /// </para>
205         /// <para></para>
206         /// </summary>
207         /// <param name="first">
208         /// <para>The first.</para>
209         /// <para></para>
210         /// </param>
```

```
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessThan(ulong first, ulong second) => first < second;

        /// <summary>
        /// <para>
        /// Determines whether this instance less or equal than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThan(ulong first, ulong second) => first > second;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater or equal than.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;

        /// <summary>
        /// <para>
        /// Gets the zero.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetZero() => 0UL;

        /// <summary>
        /// <para>
        /// Gets the one.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetOne() => 1UL;

        /// <summary>
        /// <para>
        /// Converts the to int 64 using the specified value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The long</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override long ConvertToInt64(ulong value) => (long)value;

        /// <summary>
        /// <para>
        /// Converts the to address using the specified value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong ConvertToAddress(long value) => (ulong)value;

        /// <summary>
        /// <para>
        /// Adds the first.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Add(ulong first, ulong second) => first + second;

        /// <summary>
```

```
        /// <para>
        /// Subtracts the first.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="first">
        /// <para>The first.</para>
        /// <para></para>
        /// </param>
        /// <param name="second">
        /// <para>The second.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Subtract(ulong first, ulong second) => first - second;

        /// <summary>
        /// <para>
        /// Increments the link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Increment(ulong link) => ++link;

        /// <summary>
        /// <para>
        /// Decrements the link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Decrement(ulong link) => --link;
    }
}
```

1.111 ./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
{
    /// <summary>
    /// <para>
    /// Represents the int 64 unused links list methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="UnusedLinksListMethods{ulong}"/>
    public unsafe class UInt64UnusedLinksListMethods : UnusedLinksListMethods<ulong>
    {
        private readonly RawLink<ulong>* _links;
        private readonly LinksHeader<ulong>* _header;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt64UnusedLinksListMethods"/> instance.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="header">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64UnusedLinksListMethods(RawLink<ulong>* links, LinksHeader<ulong>* header)
            : base((byte*)links, (byte*)header)
        {
            _links = links;
            _header = header;
        }

        /// <summary>
        /// <para>
        /// Gets the link reference using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A ref raw link of ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref _links[link];

        /// <summary>
        /// <para>
        /// Gets the header reference.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A ref links header of ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref LinksHeader<ulong> GetHeaderReference() => ref *_header;
    }
}
```

1.112 ./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
{
    /// <summary>
    /// <para>
    /// Represents the properties operator.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
    /// <seealso cref="IProperties{TLinkAddress, TLinkAddress, TLinkAddress}"/>
    public class PropertiesOperator<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
    ↪  IProperties<TLinkAddress, TLinkAddress, TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="PropertiesOperator"/> instance.
        /// </para>
        /// <para></para>
```

```
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public PropertiesOperator(ILinks<TLinkAddress> links) : base(links) { }

        /// <summary>
        /// <para>
        /// Gets the value using the specified object.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="@object">
        /// <para>The object.</para>
        /// <para></para>
        /// </param>
        /// <param name="property">
        /// <para>The property.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress GetValue(TLinkAddress @object, TLinkAddress 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<TLinkAddress>(any, objectProperty, any);
            var valueLink = links.SingleOrDefault(query);
            if (valueLink == null)
            {
                return default;
            }
            return links.GetTarget(valueLink[constants.IndexPart]);
        }

        /// <summary>
        /// <para>
        /// Sets the value using the specified object.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="@object">
        /// <para>The object.</para>
        /// <para></para>
        /// </param>
        /// <param name="property">
        /// <para>The property.</para>
        /// <para></para>
        /// </param>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void SetValue(TLinkAddress @object, TLinkAddress property, TLinkAddress value)
        {
            var links = _links;
            var objectProperty = links.GetOrCreate(@object, property);
            links.DeleteMany(links.AllIndices(links.Constants.Any, objectProperty));
            links.GetOrCreate(objectProperty, value);
        }
    }
}
```

1.113 ./csharp/Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs

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

```
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.PropertyOperators
8   {
9       /// <summary>
10      /// <para>
11      /// Represents the property operator.
12      /// </para>
13      /// <para></para>
14      /// </summary>
15      /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
16      /// <seealso cref="IProperty{TLinkAddress, TLinkAddress}"/>
17      public class PropertyOperator<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
            ↪ IProperty<TLinkAddress, TLinkAddress>
18      {
19          private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
                ↪ EqualityComparer<TLinkAddress>.Default;
20          private readonly TLinkAddress _propertyMarker;
21          private readonly TLinkAddress _propertyValueMarker;
22
23          /// <summary>
24          /// <para>
25          /// Initializes a new <see cref="PropertyOperator"/> instance.
26          /// </para>
27          /// <para></para>
28          /// </summary>
29          /// <param name="links">
30          /// <para>A links.</para>
31          /// <para></para>
32          /// </param>
33          /// <param name="propertyMarker">
34          /// <para>A property marker.</para>
35          /// <para></para>
36          /// </param>
37          /// <param name="propertyValueMarker">
38          /// <para>A property value marker.</para>
39          /// <para></para>
40          /// </param>
41          [MethodImpl(MethodImplOptions.AggressiveInlining)]
42          public PropertyOperator(ILinks<TLinkAddress> links, TLinkAddress propertyMarker,
                ↪ TLinkAddress propertyValueMarker) : base(links)
43          {
44              _propertyMarker = propertyMarker;
45              _propertyValueMarker = propertyValueMarker;
46          }
47
48          /// <summary>
49          /// <para>
50          /// Gets the link.
51          /// </para>
52          /// <para></para>
53          /// </summary>
54          /// <param name="link">
55          /// <para>The link.</para>
56          /// <para></para>
57          /// </param>
58          /// <returns>
59          /// <para>The link</para>
60          /// <para></para>
61          /// </returns>
62          [MethodImpl(MethodImplOptions.AggressiveInlining)]
63          public TLinkAddress Get(TLinkAddress link)
64          {
65              var property = _links.SearchOrDefault(link, _propertyMarker);
66              return GetValue(GetContainer(property));
67          }
68          [MethodImpl(MethodImplOptions.AggressiveInlining)]
69          private TLinkAddress GetContainer(TLinkAddress property)
70          {
71              var valueContainer = default(TLinkAddress);
72              if (_equalityComparer.Equals(property, default))
73              {
74                  return valueContainer;
75              }
76              var links = _links;
77              var constants = links.Constants;
78              var countinueConstant = constants.Continue;
79              var breakConstant = constants.Break;
```

```
            var anyConstant = constants.Any;
            var query = new Link<TLinkAddress>(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 TLinkAddress GetValue(TLinkAddress container) =>
        ↪  _equalityComparer.Equals(container, default) ? default : _links.GetTarget(container);

        /// <summary>
        /// <para>
        /// Sets the link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Set(TLinkAddress link, TLinkAddress 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);
            }
        }
    }
}
```

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

```
using System.Collections;
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
{
    /// <summary>
    /// <para>
    /// Represents the stack.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
    /// <seealso cref="IStack{TLinkAddress}"/>
    public class Stack<TLinkAddress> : LinksOperatorBase<TLinkAddress>, IStack<TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;
        private readonly TLinkAddress _stack;

        /// <summary>
        /// <para>
        /// Gets the is empty value.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        public bool IsEmpty
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _equalityComparer.Equals(Peek(), _stack);
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Stack"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="stack">
        /// <para>A stack.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Stack(ILinks<TLinkAddress> links, TLinkAddress stack) : base(links) => _stack =
        ↪  stack;
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private TLinkAddress GetStackMarker() => _links.GetSource(_stack);
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private TLinkAddress GetTop() => _links.GetTarget(_stack);

        /// <summary>
        /// <para>
        /// Peeks this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Peek() => _links.GetTarget(GetTop());

        /// <summary>
        /// <para>
        /// Pops this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The element.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress 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;
        }

        /// <summary>
        /// <para>
        /// Pushes the element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
104         public void Push(TLinkAddress element) => _links.Update(_stack, GetStackMarker(),
            ↪ _links.GetOrCreate(GetTop(), element));
105     }
106 }
```

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

```
1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Stacks
6  {
7      /// <summary>
8      /// <para>
9      /// Represents the stack extensions.
10     /// </para>
11     /// <para></para>
12     /// </summary>
13     public static class StackExtensions
14     {
15         /// <summary>
16         /// <para>
17         /// Creates the stack using the specified links.
18         /// </para>
19         /// <para></para>
20         /// </summary>
21         /// <typeparam name="TLinkAddress">
22         /// <para>The link.</para>
23         /// <para></para>
24         /// </typeparam>
25         /// <param name="links">
26         /// <para>The links.</para>
27         /// <para></para>
28         /// </param>
29         /// <param name="stackMarker">
30         /// <para>The stack marker.</para>
31         /// <para></para>
32         /// </param>
33         /// <returns>
34         /// <para>The stack.</para>
35         /// <para></para>
36         /// </returns>
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         public static TLinkAddress CreateStack<TLinkAddress>(this ILinks<TLinkAddress> links,
           ↪ TLinkAddress stackMarker)
39         {
40             var stackPoint = links.CreatePoint();
41             var stack = links.Update(stackPoint, stackMarker, stackPoint);
42             return stack;
43         }
44     }
45 }
```

1.116 ./csharp/Platform.Data.Doublets/SynchronizedLinks.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Data.Doublets;
5  using Platform.Delegates;
6  using Platform.Threading.Synchronization;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets
11 {
12     /// <remarks>
13     /// TODO: Autogeneration of synchronized wrapper (decorator).
14     /// TODO: Try to unfold code of each method using IL generation for performance improvements.
15     /// TODO: Or even to unfold multiple layers of implementations.
16     /// </remarks>
17     public class SynchronizedLinks<TLinkAddress> : ISynchronizedLinks<TLinkAddress>
18     {
19         /// <summary>
20         /// <para>
21         /// Gets the constants value.
22         /// </para>
23         /// <para></para>
24         /// </summary>
25         public LinksConstants<TLinkAddress> Constants
```

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

        /// <summary>
        /// <para>
        /// Gets the sync root value.
        /// </para>
        /// <para></para>
        /// </summary>
        public ISynchronization SyncRoot
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        /// <summary>
        /// <para>
        /// Gets the sync value.
        /// </para>
        /// <para></para>
        /// </summary>
        public ILinks<TLinkAddress> Sync
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        /// <summary>
        /// <para>
        /// Gets the unsync value.
        /// </para>
        /// <para></para>
        /// </summary>
        public ILinks<TLinkAddress> Unsync
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="SynchronizedLinks"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SynchronizedLinks(ILinks<TLinkAddress> links) : this(new
        ↪  ReaderWriterLockSynchronization(), links) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="SynchronizedLinks"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="synchronization">
        /// <para>A synchronization.</para>
        /// <para></para>
        /// </param>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SynchronizedLinks(ISynchronization synchronization, ILinks<TLinkAddress> links)
        {
            SyncRoot = synchronization;
            Sync = this;
            Unsync = links;
            Constants = links.Constants;
        }

        /// <summary>
```

```
        /// <para>
        /// Counts the restriction.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restriction">
        /// <para>The restriction.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link address</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Count(IList<TLinkAddress>? restriction) =>
        ↪  SyncRoot.ExecuteReadOperation(restriction, Unsync.Count);

        /// <summary>
        /// <para>
        /// Eaches the handler.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="handler">
        /// <para>The handler.</para>
        /// <para></para>
        /// </param>
        /// <param name="restriction">
        /// <para>The substitution.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link address</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Each(IList<TLinkAddress>? restriction, ReadHandler<TLinkAddress>?
        ↪  handler) => SyncRoot.ExecuteReadOperation(restriction, handler, Unsync.Each);

        /// <summary>
        /// <para>
        /// Creates the substitution.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="substitution">
        /// <para>The substitution.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link address</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Create(IList<TLinkAddress>? substitution,
        ↪  WriteHandler<TLinkAddress>? handler) => SyncRoot.ExecuteWriteOperation(substitution,
        ↪  handler, Unsync.Create);

        /// <summary>
        /// <para>
        /// Updates the substitution.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restriction">
        /// <para>The substitution.</para>
        /// <para></para>
        /// </param>
        /// <param name="substitution">
        /// <para>The substitution.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link address</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
177         public TLinkAddress Update(IList<TLinkAddress>? restriction, IList<TLinkAddress>?
            ↪  substitution, WriteHandler<TLinkAddress>? handler) =>
            ↪  SyncRoot.ExecuteWriteOperation(restriction, substitution, handler, Unsync.Update);
178
179         /// <summary>
180         /// <para>
181         /// Deletes the substitution.
182         /// </para>
183         /// <para></para>
184         /// </summary>
185         /// <param name="restriction">
186         /// <para>The substitution.</para>
187         /// <para></para>
188         /// </param>
189         [MethodImpl(MethodImplOptions.AggressiveInlining)]
190         public TLinkAddress Delete(IList<TLinkAddress>? restriction, WriteHandler<TLinkAddress>?
            ↪  handler) => SyncRoot.ExecuteWriteOperation(restriction, handler, Unsync.Delete);
191
192         //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
            ↪  IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
193         //{
194         //    if (restriction != null && substitution != null &&
            ↪  !substitution.EqualTo(restriction))
195         //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
            ↪  substitution, substitutedHandler, Unsync.Trigger);
196
197         //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
            ↪  substitutedHandler, Unsync.Trigger);
198         //}
199     }
200 }
```

1.117 ./csharp/Platform.Data.Doublets/UInt64LinksExtensions.cs

```
1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Singletons;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets
10 {
11     /// <summary>
12     /// <para>
13     /// Represents the int 64 links extensions.
14     /// </para>
15     /// <para></para>
16     /// </summary>
17     public static class UInt64LinksExtensions
18     {
19         /// <summary>
20         /// <para>
21         /// The instance.
22         /// </para>
23         /// <para></para>
24         /// </summary>
25         public static readonly LinksConstants<ulong> Constants =
            ↪  Default<LinksConstants<ulong>>.Instance;
26
27         /// <summary>
28         /// <para>
29         /// Determines whether any link is any.
30         /// </para>
31         /// <para></para>
32         /// </summary>
33         /// <param name="links">
34         /// <para>The links.</para>
35         /// <para></para>
36         /// </param>
37         /// <param name="sequence">
38         /// <para>The sequence.</para>
39         /// <para></para>
40         /// </param>
41         /// <returns>
42         /// <para>The bool</para>
43         /// <para></para>
44         /// </returns>
```

```
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
47          {
48              if (sequence == null)
49              {
50                  return false;
51              }
52              var constants = links.Constants;
53              for (var i = 0; i < sequence.Length; i++)
54              {
55                  if (sequence[i] == constants.Any)
56                  {
57                      return true;
58                  }
59              }
60              return false;
61          }
62
63          /// <summary>
64          /// <para>
65          /// Formats the structure using the specified links.
66          /// </para>
67          /// <para></para>
68          /// </summary>
69          /// <param name="links">
70          /// <para>The links.</para>
71          /// <para></para>
72          /// </param>
73          /// <param name="linkIndex">
74          /// <para>The link index.</para>
75          /// <para></para>
76          /// </param>
77          /// <param name="isElement">
78          /// <para>The is element.</para>
79          /// <para></para>
80          /// </param>
81          /// <param name="renderIndex">
82          /// <para>The render index.</para>
83          /// <para></para>
84          /// </param>
85          /// <param name="renderDebug">
86          /// <para>The render debug.</para>
87          /// <para></para>
88          /// </param>
89          /// <returns>
90          /// <para>The string</para>
91          /// <para></para>
92          /// </returns>
93          [MethodImpl(MethodImplOptions.AggressiveInlining)]
94          public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
            ↪  Func<Link<ulong>, bool> isElement, bool renderIndex = false, bool renderDebug =
            ↪  false)
95          {
96              var sb = new StringBuilder();
97              var visited = new HashSet<ulong>();
98              links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
                ↪  innerSb.Append(link.Index), renderIndex, renderDebug);
99              return sb.ToString();
100         }
101
102         /// <summary>
103         /// <para>
104         /// Formats the structure using the specified links.
105         /// </para>
106         /// <para></para>
107         /// </summary>
108         /// <param name="links">
109         /// <para>The links.</para>
110         /// <para></para>
111         /// </param>
112         /// <param name="linkIndex">
113         /// <para>The link index.</para>
114         /// <para></para>
115         /// </param>
116         /// <param name="isElement">
117         /// <para>The is element.</para>
118         /// <para></para>
119         /// </param>
```

```
120         /// <param name="appendElement">
121         /// <para>The append element.</para>
122         /// <para></para>
123         /// </param>
124         /// <param name="renderIndex">
125         /// <para>The render index.</para>
126         /// <para></para>
127         /// </param>
128         /// <param name="renderDebug">
129         /// <para>The render debug.</para>
130         /// <para></para>
131         /// </param>
132         /// <returns>
133         /// <para>The string</para>
134         /// <para></para>
135         /// </returns>
136         [MethodImpl(MethodImplOptions.AggressiveInlining)]
137         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)
138         {
139             var sb = new StringBuilder();
140             var visited = new HashSet<ulong>();
141             links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
                  ↪ renderDebug);
142             return sb.ToString();
143         }
144
145         /// <summary>
146         /// <para>
147         /// Appends the structure using the specified links.
148         /// </para>
149         /// <para></para>
150         /// </summary>
151         /// <param name="links">
152         /// <para>The links.</para>
153         /// <para></para>
154         /// </param>
155         /// <param name="sb">
156         /// <para>The sb.</para>
157         /// <para></para>
158         /// </param>
159         /// <param name="visited">
160         /// <para>The visited.</para>
161         /// <para></para>
162         /// </param>
163         /// <param name="linkIndex">
164         /// <para>The link index.</para>
165         /// <para></para>
166         /// </param>
167         /// <param name="isElement">
168         /// <para>The is element.</para>
169         /// <para></para>
170         /// </param>
171         /// <param name="appendElement">
172         /// <para>The append element.</para>
173         /// <para></para>
174         /// </param>
175         /// <param name="renderIndex">
176         /// <para>The render index.</para>
177         /// <para></para>
178         /// </param>
179         /// <param name="renderDebug">
180         /// <para>The render debug.</para>
181         /// <para></para>
182         /// </param>
183         /// <exception cref="ArgumentNullException">
184         /// <para></para>
185         /// <para></para>
186         /// </exception>
187         [MethodImpl(MethodImplOptions.AggressiveInlining)]
188         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)
189         {
190             if (sb == null)
```

```
191             {
192                 throw new ArgumentNullException(nameof(sb));
193             }
194             if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
                ↪  Constants.Itself)
195             {
196                 return;
197             }
198             if (links.Exists(linkIndex))
199             {
200                 if (visited.Add(linkIndex))
201                 {
202                     sb.Append('(');
203                     var link = new Link<ulong>(links.GetLink(linkIndex));
204                     if (renderIndex)
205                     {
206                         sb.Append(link.Index);
207                         sb.Append(':');
208                     }
209                     if (link.Source == link.Index)
210                     {
211                         sb.Append(link.Index);
212                     }
213                     else
214                     {
215                         var source = new Link<ulong>(links.GetLink(link.Source));
216                         if (isElement(source))
217                         {
218                             appendElement(sb, source);
219                         }
220                         else
221                         {
222                             links.AppendStructure(sb, visited, source.Index, isElement,
                                ↪  appendElement, renderIndex);
223                         }
224                     }
225                     sb.Append(' ');
226                     if (link.Target == link.Index)
227                     {
228                         sb.Append(link.Index);
229                     }
230                     else
231                     {
232                         var target = new Link<ulong>(links.GetLink(link.Target));
233                         if (isElement(target))
234                         {
235                             appendElement(sb, target);
236                         }
237                         else
238                         {
239                             links.AppendStructure(sb, visited, target.Index, isElement,
                                ↪  appendElement, renderIndex);
240                         }
241                     }
242                     sb.Append(')');
243                 }
244                 else
245                 {
246                     if (renderDebug)
247                     {
248                         sb.Append('*');
249                     }
250                     sb.Append(linkIndex);
251                 }
252             }
253             else
254             {
255                 if (renderDebug)
256                 {
257                     sb.Append('~');
258                 }
259                 sb.Append(linkIndex);
260             }
261         }
262     }
263 }
```

```
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.Delegates;
using Platform.Exceptions;

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

namespace Platform.Data.Doublets
{
    /// <summary>
    /// <para>
    /// Represents the int 64 links transactions layer.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksDisposableDecoratorBase{ulong}"/>
    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;amp; TransactionIdCombined;
        ///         }
        ///     }
        ///
        ///     public UniqueTimestamp Timestamp
        ///     {
        ///         get
        ///         {
        ///             return (UniqueTimestamp)mask &amp;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>
        {
            /// <summary>
            /// <para>
            /// The size.
            /// </para>
            /// <para></para>
            /// </summary>
            public static readonly long Size = Structure<Transition>.Size;

            /// <summary>
            /// <para>
            /// The transaction id.
            /// </para>
            /// <para></para>
            /// </summary>
            public readonly ulong TransactionId;
            /// <summary>
            /// <para>
            /// The before.
            /// </para>
            /// <para></para>
            /// </summary>
            public readonly Link<ulong> Before;
            /// <summary>
            /// <para>
            /// The after.
            /// </para>
            /// <para></para>
            /// </summary>
            public readonly Link<ulong> After;
            /// <summary>
            /// <para>
            /// The timestamp.
            /// </para>
            /// <para></para>
            /// </summary>
            public readonly Timestamp Timestamp;

            /// <summary>
            /// <para>
            /// Initializes a new <see cref="Transition"/> instance.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="uniqueTimestampFactory">
            /// <para>A unique timestamp factory.</para>
            /// <para></para>
            /// </param>
            /// <param name="transactionId">
            /// <para>A transaction id.</para>
            /// <para></para>
            /// </param>
            /// <param name="before">
            /// <para>A before.</para>
            /// <para></para>
            /// </param>
            /// <param name="after">
            /// <para>A after.</para>
            /// <para></para>
            /// </param>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
            ↪  transactionId, Link<ulong> before, Link<ulong> after)
```

```
        {
            TransactionId = transactionId;
            Before = before;
            After = after;
            Timestamp = uniqueTimestampFactory.Create();
        }

        public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId, IList<ulong> before, IList<ulong> after) : this(uniqueTimestampFactory, transactionId, new Link<ulong>(before), new Link<ulong>(after)) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Transition"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="uniqueTimestampFactory">
        /// <para>A unique timestamp factory.</para>
        /// <para></para>
        /// </param>
        /// <param name="transactionId">
        /// <para>A transaction id.</para>
        /// <para></para>
        /// </param>
        /// <param name="before">
        /// <para>A before.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId, Link<ulong> before) : this(uniqueTimestampFactory, transactionId, before, default) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Transition"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="uniqueTimestampFactory">
        /// <para>A unique timestamp factory.</para>
        /// <para></para>
        /// </param>
        /// <param name="transactionId">
        /// <para>A transaction id.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId) : this(uniqueTimestampFactory, transactionId, default, default) { }

        /// <summary>
        /// <para>
        /// Returns the string.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override string ToString() => $"{Timestamp} {TransactionId}: {Before} => {After}";

        /// <summary>
        /// <para>
        /// Determines whether this instance equals.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="obj">
        /// <para>The obj.</para>
        /// <para></para>
        /// </param>
```

```
226             /// <returns>
227             /// <para>The bool</para>
228             /// <para></para>
229             /// </returns>
230             [MethodImpl(MethodImplOptions.AggressiveInlining)]
231             public override bool Equals(object obj) => obj is Transition transition ?
                ↪ Equals(transition) : false;
232
233             /// <summary>
234             /// <para>
235             /// Gets the hash code.
236             /// </para>
237             /// <para></para>
238             /// </summary>
239             /// <returns>
240             /// <para>The int</para>
241             /// <para></para>
242             /// </returns>
243             [MethodImpl(MethodImplOptions.AggressiveInlining)]
244             public override int GetHashCode() => (TransactionId, Before, After,
                ↪ Timestamp).GetHashCode();
245
246             /// <summary>
247             /// <para>
248             /// Determines whether this instance equals.
249             /// </para>
250             /// <para></para>
251             /// </summary>
252             /// <param name="other">
253             /// <para>The other.</para>
254             /// <para></para>
255             /// </param>
256             /// <returns>
257             /// <para>The bool</para>
258             /// <para></para>
259             /// </returns>
260             [MethodImpl(MethodImplOptions.AggressiveInlining)]
261             public bool Equals(Transition other) => TransactionId == other.TransactionId &&
                ↪ Before == other.Before && After == other.After && Timestamp == other.Timestamp;
262
263             [MethodImpl(MethodImplOptions.AggressiveInlining)]
264             public static bool operator ==(Transition left, Transition right) =>
                ↪ left.Equals(right);
265
266             [MethodImpl(MethodImplOptions.AggressiveInlining)]
267             public static bool operator !=(Transition left, Transition right) => !(left ==
                ↪ right);
268         }
269
270         /// <remarks>
271         /// Другие варианты реализации транзакций (атомарности):
272         ///     1. Разделение хранения значения связи ((Source Target) или (Source Linker
            ↪ Target)) и индексов.
273         ///     2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
            ↪ потребуется решить вопрос
274         ///        со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
            ↪ пересечениями идентификаторов.
275         ///
276         /// Где хранить промежуточный список транзакций?
277         ///
278         /// В оперативной памяти:
279         ///  Минусы:
280         ///     1. Может усложнить систему, если она будет функционировать самостоятельно,
281         ///     так как нужно отдельно выделять память под список трансформаций.
282         ///     2. Выделенной оперативной памяти может не хватить, в том случае,
283         ///     если транзакция использует слишком много трансформаций.
284         ///         -> Можно использовать жёсткий диск для слишком длинных транзакций.
285         ///         -> Максимальный размер списка трансформаций можно ограничить / задать
            ↪ константой.
286         ///     3. При подтверждении транзакции (Commit) все трансформации записываются разом
            ↪ создавая задержку.
287         ///
288         /// На жёстком диске:
289         ///  Минусы:
290         ///     1. Длительный отклик, на запись каждой трансформации.
291         ///     2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
292         ///         -> Это может решаться упаковкой/исключением дублирующих операций.
293         ///         -> Также это может решаться тем, что короткие транзакции вообще
```

```
294         ///             не будут записываться в случае отката.
295         ///     3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
            ↪  операции (трансформации)
296         ///        будут записаны в лог.
297         ///
298         /// </remarks>
299         public class Transaction : DisposableBase
300         {
301             private readonly Queue<Transition> _transitions;
302             private readonly UInt64LinksTransactionsLayer _layer;
303             /// <summary>
304             /// <para>
305             /// Gets or sets the is committed value.
306             /// </para>
307             /// <para></para>
308             /// </summary>
309             public bool IsCommitted { get; private set; }
310             /// <summary>
311             /// <para>
312             /// Gets or sets the is reverted value.
313             /// </para>
314             /// <para></para>
315             /// </summary>
316             public bool IsReverted { get; private set; }
317 
318             /// <summary>
319             /// <para>
320             /// Initializes a new <see cref="Transaction"/> instance.
321             /// </para>
322             /// <para></para>
323             /// </summary>
324             /// <param name="layer">
325             /// <para>A layer.</para>
326             /// <para></para>
327             /// </param>
328             /// <exception cref="NotSupportedException">
329             /// <para>Nested transactions not supported.</para>
330             /// <para></para>
331             /// </exception>
332             [MethodImpl(MethodImplOptions.AggressiveInlining)]
333             public Transaction(UInt64LinksTransactionsLayer layer)
334             {
335                 _layer = layer;
336                 if (_layer._currentTransactionId != 0)
337                 {
338                     throw new NotSupportedException("Nested transactions not supported.");
339                 }
340                 IsCommitted = false;
341                 IsReverted = false;
342                 _transitions = new Queue<Transition>();
343                 SetCurrentTransaction(layer, this);
344             }
345 
346             /// <summary>
347             /// <para>
348             /// Commits this instance.
349             /// </para>
350             /// <para></para>
351             /// </summary>
352             [MethodImpl(MethodImplOptions.AggressiveInlining)]
353             public void Commit()
354             {
355                 EnsureTransactionAllowsWriteOperations(this);
356                 while (_transitions.Count > 0)
357                 {
358                     var transition = _transitions.Dequeue();
359                     _layer._transitions.Enqueue(transition);
360                 }
361                 _layer._lastCommitedTransactionId = _layer._currentTransactionId;
362                 IsCommitted = true;
363             }
364             [MethodImpl(MethodImplOptions.AggressiveInlining)]
365             private void Revert()
366             {
367                 EnsureTransactionAllowsWriteOperations(this);
368                 var transitionsToRevert = new Transition[_transitions.Count];
369                 _transitions.CopyTo(transitionsToRevert, 0);
370                 for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
```

```
                {
                    _layer.RevertTransition(transitionsToRevert[i]);
                }
                IsReverted = true;
            }

            /// <summary>
            /// <para>
            /// Sets the current transaction using the specified layer.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="layer">
            /// <para>The layer.</para>
            /// <para></para>
            /// </param>
            /// <param name="transaction">
            /// <para>The transaction.</para>
            /// <para></para>
            /// </param>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
            ↪  Transaction transaction)
            {
                layer._currentTransactionId = layer._lastCommitedTransactionId + 1;
                layer._currentTransactionTransitions = transaction._transitions;
                layer._currentTransaction = transaction;
            }

            /// <summary>
            /// <para>
            /// Ensures the transaction allows write operations using the specified transaction.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="transaction">
            /// <para>The transaction.</para>
            /// <para></para>
            /// </param>
            /// <exception cref="InvalidOperationException">
            /// <para>Transation is commited.</para>
            /// <para></para>
            /// </exception>
            /// <exception cref="InvalidOperationException">
            /// <para>Transation is reverted.</para>
            /// <para></para>
            /// </exception>
            [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.");
                }
            }

            /// <summary>
            /// <para>
            /// Disposes the manual.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="manual">
            /// <para>The manual.</para>
            /// <para></para>
            /// </param>
            /// <param name="wasDisposed">
            /// <para>The was disposed.</para>
            /// <para></para>
            /// </param>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override void Dispose(bool manual, bool wasDisposed)
            {
                if (!wasDisposed && _layer != null && !_layer.Disposable.IsDisposed)
```

```
            {
                if (!IsCommitted && !IsReverted)
                {
                    Revert();
                }
                _layer.ResetCurrentTransation();
            }
        }
    }

    /// <summary>
    /// <para>
    /// The from seconds.
    /// </para>
    /// <para></para>
    /// </summary>
    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;

    /// <summary>
    /// <para>
    /// Initializes a new <see cref="UInt64LinksTransactionsLayer"/> instance.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <param name="links">
    /// <para>A links.</para>
    /// <para></para>
    /// </param>
    /// <param name="logAddress">
    /// <para>A log address.</para>
    /// <para></para>
    /// </param>
    /// <exception cref="ArgumentNullException">
    /// <para></para>
    /// <para></para>
    /// </exception>
    /// <exception cref="NotSupportedException">
    /// <para>Database is damaged, autorecovery is not supported yet.</para>
    /// <para></para>
    /// </exception>
    [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();
        }

        /// <summary>
        /// <para>
        /// Gets the link value using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A list of ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IList<ulong> GetLinkValue(ulong link) => _links.GetLink(link);

        /// <summary>
        /// <para>
        /// Creates the substitution.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="substitution">
        /// <para>The substitution.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The created link index.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override ulong Create(IList<ulong> substitution, WriteHandler<ulong> handler)
        {
            return _links.Create(new Link<ulong>(), (before, after) =>
            {
                CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                ↪  new Link<ulong>(before), new Link<ulong>(after)));
                return handler(before, after);
            });
        }

        /// <summary>
        /// <para>
        /// Updates the substitution.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restriction">
        /// <para>The substitution.</para>
        /// <para></para>
        /// </param>
        /// <param name="substitution">
        /// <para>The substitution.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link index.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override ulong Update(IList<ulong> restriction, IList<ulong> substitution,
        ↪  WriteHandler<ulong> handler)
        {
            return _links.Update(restriction, substitution, (before, after) =>
                {
                    CommitTransition(new Transition(_uniqueTimestampFactory,
                    ↪  _currentTransactionId, new Link<ulong>(before), new Link<ulong>(after)));
                    return handler(before, after);
                }
```

```
600             );
601         }
602 
603         /// <summary>
604         /// <para>
605         /// Deletes the substitution.
606         /// </para>
607         /// <para></para>
608         /// </summary>
609         /// <param name="restriction">
610         /// <para>The substitution.</para>
611         /// <para></para>
612         /// </param>
613         [MethodImpl(MethodImplOptions.AggressiveInlining)]
614         public override ulong Delete(IList<ulong> restriction, WriteHandler<ulong> handler)
615         {
616             var link = restriction[_constants.IndexPart];
617             return _links.Delete(restriction, (before, after) =>
618             {
619                 CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                    ↪  before, after));
620                 return handler(before, after);
621             });
622         }
623         [MethodImpl(MethodImplOptions.AggressiveInlining)]
624         private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
            ↪  _transitions;
625         [MethodImpl(MethodImplOptions.AggressiveInlining)]
626         private void CommitTransition(Transition transition)
627         {
628             if (_currentTransaction != null)
629             {
630                 Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
631             }
632             var transitions = GetCurrentTransitions();
633             transitions.Enqueue(transition);
634         }
635         [MethodImpl(MethodImplOptions.AggressiveInlining)]
636         private void RevertTransition(Transition transition)
637         {
638             if (transition.After.IsNull()) // Revert Deletion with Creation
639             {
640                 _links.Create();
641             }
642             else if (transition.Before.IsNull()) // Revert Creation with Deletion
643             {
644                 _links.Delete(transition.After.Index);
645             }
646             else // Revert Update
647             {
648                 _links.Update(new[] { transition.After.Index, transition.Before.Source,
                    ↪  transition.Before.Target });
649             }
650         }
651         [MethodImpl(MethodImplOptions.AggressiveInlining)]
652         private void ResetCurrentTransation()
653         {
654             _currentTransactionId = 0;
655             _currentTransactionTransitions = null;
656             _currentTransaction = null;
657         }
658         [MethodImpl(MethodImplOptions.AggressiveInlining)]
659         private void PushTransitions()
660         {
661             if (_log == null || _transitions == null)
662             {
663                 return;
664             }
665             for (var i = 0; i < _transitions.Count; i++)
666             {
667                 var transition = _transitions.Dequeue();
668 
669                 _log.Write(transition);
670                 _lastCommitedTransition = transition;
671             }
672         }
673         [MethodImpl(MethodImplOptions.AggressiveInlining)]
674         private void TransitionsPusher()
```

```
675         {
676             while (!Disposable.IsDisposed && _transitionsPusher != null)
677             {
678                 Thread.Sleep(DefaultPushDelay);
679                 PushTransitions();
680             }
681         }
682 
683         /// <summary>
684         /// <para>
685         /// Begins the transaction.
686         /// </para>
687         /// <para></para>
688         /// </summary>
689         /// <returns>
690         /// <para>The transaction</para>
691         /// <para></para>
692         /// </returns>
693         [MethodImpl(MethodImplOptions.AggressiveInlining)]
694         public Transaction BeginTransaction() => new Transaction(this);
695         [MethodImpl(MethodImplOptions.AggressiveInlining)]
696         private void DisposeTransitions()
697         {
698             try
699             {
700                 var pusher = _transitionsPusher;
701                 if (pusher != null)
702                 {
703                     _transitionsPusher = null;
704                     pusher.Wait();
705                 }
706                 if (_transitions != null)
707                 {
708                     PushTransitions();
709                 }
710                 _log.DisposeIfPossible();
711                 FileHelpers.WriteFirst(_logAddress, _lastCommitedTransition);
712             }
713             catch (Exception ex)
714             {
715                 ex.Ignore();
716             }
717         }
718 
719         #region DisposalBase
720 
721         /// <summary>
722         /// <para>
723         /// Disposes the manual.
724         /// </para>
725         /// <para></para>
726         /// </summary>
727         /// <param name="manual">
728         /// <para>The manual.</para>
729         /// <para></para>
730         /// </param>
731         /// <param name="wasDisposed">
732         /// <para>The was disposed.</para>
733         /// <para></para>
734         /// </param>
735         [MethodImpl(MethodImplOptions.AggressiveInlining)]
736         protected override void Dispose(bool manual, bool wasDisposed)
737         {
738             if (!wasDisposed)
739             {
740                 DisposeTransitions();
741             }
742             base.Dispose(manual, wasDisposed);
743         }
744 
745         #endregion
746     }
747 }
```

1.119 ./csharp/Platform.Data.Doublets.Tests/GenericLinksTests.cs

```
1  using System;
2  using System.IO;
3  using Platform.Data.Doublets.Decorators;
4  using Xunit;
```

```

using Platform.Memory;

using Platform.Data.Doublets.Memory.United.Generic;

namespace Platform.Data.Doublets.Tests
{
    public 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().TestMultipleRandomCreationsAndDeletions(16)); // Cannot use more because current implementation of tree cuts out 5 bits from the address space.
            Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultipleRandomCreationsAndDeletions(100));
            Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultipleRandomCreationsAndDeletions(100));
            Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultipleRandomCreationsAndDeletions(100));
        }
        private static void Using<TLinkAddress>(Action<ILinks<TLinkAddress>> action) where TLinkAddress : struct
        {
            var unitedMemoryLinks = new UnitedMemoryLinks<TLinkAddress>(new HeapResizableDirectMemory());
            using (var logFile = File.Open("linksLogger.txt", FileMode.Create, FileAccess.Write))
            {
                LoggingDecorator<TLinkAddress> links = new(unitedMemoryLinks, logFile);
                action(links);
            }

            File.Delete("db.links");
            using var ffiLinks = new FFI.UnitedMemoryLinks<TLinkAddress>("db.links");
            action(ffiLinks);
        }
    }
}
```

1.120 ./csharp/Platform.Data.Doublets.Tests/ILinksBasicTests.cs

```

using System.IO;
using Platform.Data.Doublets.Memory.United.Generic;
using Platform.Memory;
using Xunit;


namespace Platform.Data.Doublets.Tests
{
    public static class ILinksBasicTests
    {
        [Fact]
        public static void DeleteAllUsages()
        {
            var mem = new HeapResizableDirectMemory();
            var links = new UnitedMemoryLinks<uint>(mem);

            var root = links.CreatePoint();

            var a = links.CreatePoint();
            var b = links.CreatePoint();
```

```
21
22              links.CreateAndUpdate(a, root);
23              links.CreateAndUpdate(b, root);
24
25              Assert.Equal(5U, links.Count());
26
27              links.DeleteAllUsages(root);
28
29              Assert.Equal(3U, links.Count());
30          }
31
32          [Fact]
33          public static void FfiDeleteAllUsages()
34          {
35              File.Delete("db.links");
36              var links = new FFI.UnitedMemoryLinks<uint>("db.links");
37
38              var root = links.CreatePoint();
39
40              var a = links.CreatePoint();
41              var b = links.CreatePoint();
42
43              links.CreateAndUpdate(a, root);
44              links.CreateAndUpdate(b, root);
45
46              Assert.Equal(5U, links.Count());
47
48              links.DeleteAllUsages(root);
49
50              Assert.Equal(3U, links.Count());
51          }
52      }
53  }
```

1.121 ./csharp/Platform.Data.Doublets.Tests/LinksConstantsTests.cs

```
1   using Xunit;
2
3   namespace Platform.Data.Doublets.Tests
4   {
5       public static class LinksConstantsTests
6       {
7           [Fact]
8           public static void ExternalReferencesTest()
9           {
10              LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
                ↪  (long.MaxValue + 1UL, ulong.MaxValue));
11
12              //var minimum = new Hybrid<ulong>(0, isExternal: true);
13              var minimum = new Hybrid<ulong>(1, isExternal: true);
14              var maximum = new Hybrid<ulong>(long.MaxValue, isExternal: true);
15
16              Assert.True(constants.IsExternalReference(minimum));
17              Assert.True(constants.IsExternalReference(maximum));
18          }
19      }
20  }
```

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

```
1   using System.IO;
2   using Xunit;
3   using Platform.Singletons;
4   using Platform.Memory;
5   using Platform.Data.Doublets.Memory.United.Specific;
6
7   namespace Platform.Data.Doublets.Tests
8   {
9       public static class ResizableDirectMemoryLinksTests
10      {
11          private static readonly LinksConstants<ulong> _constants =
                ↪  Default<LinksConstants<ulong>>.Instance;
12
13          [Fact]
14          public static void BasicFileMappedMemoryTest()
15          {
16              var tempFilename = Path.GetTempFileName();
17              using (var memoryAdapter = new UInt64UnitedMemoryLinks(tempFilename))
18              {
19                  memoryAdapter.TestBasicMemoryOperations();
20              }
```

```
            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);
        }
    }
}
```

1.123 ./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);
            }
        }
    }
}
```

1.124 ./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<TLinkAddress>(Action<ILinks<TLinkAddress>> action) where
         ↪  TLinkAddress : struct
        {
            using (var dataMemory = new HeapResizableDirectMemory())
            using (var indexMemory = new HeapResizableDirectMemory())
            using (var memory = new SplitMemoryLinks<TLinkAddress>(dataMemory, indexMemory))
            {
                action(memory);
```

```
            }
        }
        private static void
        ↪ UsingWithExternalReferences<TLinkAddress>(Action<ILinks<TLinkAddress>> action) where
        ↪ TLinkAddress : struct
        {
            var contants = new LinksConstants<TLinkAddress>(enableExternalReferencesSupport:
            ↪ true);
            using (var dataMemory = new HeapResizableDirectMemory())
            using (var indexMemory = new HeapResizableDirectMemory())
            using (var memory = new SplitMemoryLinks<TLinkAddress>(dataMemory, indexMemory,
            ↪ SplitMemoryLinks<TLinkAddress>.DefaultLinksSizeStep, contants))
            {
                action(memory);
            }
        }
    }
}
```

1.125 ./csharp/Platform.Data.Doublets.Tests/SplitMemoryUInt32LinksTests.cs

```
using System;
using Xunit;
using Platform.Memory;
using Platform.Data.Doublets.Memory.Split.Specific;
using TLinkAddress = 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<TLinkAddress>> 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<TLinkAddress>> action)
        {
            var contants = new LinksConstants<TLinkAddress>(enableExternalReferencesSupport:
            ↪ true);
            using (var dataMemory = new HeapResizableDirectMemory())
            using (var indexMemory = new HeapResizableDirectMemory())
            using (var memory = new UInt32SplitMemoryLinks(dataMemory, indexMemory,
            ↪ UInt32SplitMemoryLinks.DefaultLinksSizeStep, contants))
            {
                action(memory);
            }
        }
    }
}
```

1.126 ./csharp/Platform.Data.Doublets.Tests/SplitMemoryUInt64LinksTests.cs

```
using System;
using Xunit;
using Platform.Memory;
using Platform.Data.Doublets.Memory.Split.Specific;
using TLinkAddress = 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<TLinkAddress>> 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<TLinkAddress>> action)
        {
            var contants = new LinksConstants<TLinkAddress>(enableExternalReferencesSupport:
            ↪  true);
            using (var dataMemory = new HeapResizableDirectMemory())
            using (var indexMemory = new HeapResizableDirectMemory())
            using (var memory = new UInt64SplitMemoryLinks(dataMemory, indexMemory,
            ↪  UInt64SplitMemoryLinks.DefaultLinksSizeStep, contants))
            {
                action(memory);
            }
        }
    }
}
```

1.127 ./csharp/Platform.Data.Doublets.Tests/TestExtensions.cs

```
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 TestMultipleCreationsAndDeletions<TLinkAddress>(this
        ↪  ILinks<TLinkAddress> links, int numberOfOperations)
        {
            for (int i = 0; i < numberOfOperations; i++)
            {
                links.Create();
            }
            for (int i = 0; i < numberOfOperations; i++)
            {
                links.Delete(links.Count());
            }
        }

        public static void TestMultipleRandomCreationsAndDeletions<TLinkAddress>(this
        ↪  ILinks<TLinkAddress> links, int maximumOperationsPerCycle)
        {
            var comparer = Comparer<TLinkAddress>.Default;
            var addressToUInt64Converter = CheckedConverter<TLinkAddress, ulong>.Default;
            var uInt64ToAddressConverter = CheckedConverter<ulong, TLinkAddress>.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);
                        TLinkAddress source = uInt64ToAddressConverter.Convert(random.NextUInt64
                        ↪  (linksAddressRange));
```

```
                        TLinkAddress target = uInt64ToAddressConverter.Convert(random.NextUInt64
                        ↪  (linksAddressRange));
                        ↪  //-V3086
                        var resultLink = links.GetOrCreate(source, target);
                        if (comparer.Compare(resultLink,
                        ↪  uInt64ToAddressConverter.Convert(linksCount)) > 0)
                        {
                            created++;
                        }
                    }
                    else
                    {
                        links.Create();
                        created++;
                    }
                }
                Assert.True(created == addressToUInt64Converter.Convert(links.Count()));
                for (var i = 0; i < N; i++)
                {
                    TLinkAddress link = uInt64ToAddressConverter.Convert((ulong)i + 1UL);
                    if (links.Exists(link))
                    {
                        links.Delete(link);
                        deleted++;
                    }
                }
                Assert.True(addressToUInt64Converter.Convert(links.Count()) == 0L);
            }
        }
    }
}
```

1.128 ./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 TLinkAddress = System.UInt64;

namespace Platform.Data.Doublets.Tests
{
    public class Uint64LinksExtensionsTests
    {
        public static ILinks<TLinkAddress> CreateLinks() => CreateLinks<TLinkAddress>(new
        ↪  Platform.IO.TemporaryFile());

        public static ILinks<TLinkAddress> CreateLinks<TLinkAddress>(string dataDBFilename)
        ↪  where TLinkAddress : struct
        {
            var linksConstants = new
            ↪  LinksConstants<TLinkAddress>(enableExternalReferencesSupport: true);
            return new UnitedMemoryLinks<TLinkAddress>(new
            ↪  FileMappedResizableDirectMemory(dataDBFilename),
            ↪  UnitedMemoryLinks<TLinkAddress>.DefaultLinksSizeStep, linksConstants,
            ↪  IndexTreeType.Default);
        }
        [Fact]
        public void FormatStructureWithExternalReferenceTest()
        {
            ILinks<TLinkAddress> links = CreateLinks();
            TLinkAddress 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<TLinkAddress> 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);
        }
    }
}
```

1.129 ./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 TLinkAddress = 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<TLinkAddress>> action)
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪ UInt32UnitedMemoryLinks>>())
            {
                action(scope.Use<ILinks<TLinkAddress>>());
            }
        }
    }
}
```

1.130 ./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 TLinkAddress = 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<TLinkAddress>> action)
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪ UInt64UnitedMemoryLinks>>())
            {
                action(scope.Use<ILinks<TLinkAddress>>());
```

```
                }
            }
        }
    }
```

Index