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 = _links.GetIndex(restriction);
            // 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 links = _links;
            var link = links.GetIndex(restriction);
            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(links.GetSource(substitution), links.GetTarget(substitution));
            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;

```

```
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksUniquenessResolver"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksUniquenessResolver(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;
            var newLinkAddress = links.SearchOrDefault(links.GetSource(substitution),
            ↪  links.GetTarget(substitution));
            if (_equalityComparer.Equals(newLinkAddress, default))
            {
                return links.Update(restriction, substitution, handler);
            }
            return ResolveAddressChangeConflict(links.GetIndex(restriction), newLinkAddress,
            ↪  handler);
        }

        /// <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 new link address.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLinkAddress ResolveAddressChangeConflict(TLinkAddress oldLinkAddress,
        ↪  TLinkAddress newLinkAddress, WriteHandler<TLinkAddress>? handler)
        {
            if (!_equalityComparer.Equals(oldLinkAddress, newLinkAddress) &&
            ↪  _links.Exists(oldLinkAddress))
            {
                return _facade.Delete(oldLinkAddress, handler);
            }
            return _links.Constants.Continue;
        }
    }
```

```
}
```

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(links.GetSource(substitution),
            ↪  links.GetTarget(substitution));
            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.
    /// </para>
```

```
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksDecoratorBase{TLinkAddress}"/>
    public class LinksUsagesValidator<TLinkAddress> : LinksDecoratorBase<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksUsagesValidator"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksUsagesValidator(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;
            links.EnsureNoUsages(links.GetIndex(restriction));
            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 links = _links;
            var link = links.GetIndex(restriction);
            links.EnsureNoUsages(link);
            return links.Delete(restriction, handler);
        }
    }
}
```

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

```
using System.Collections.Generic;
using System.IO;
using Platform.Delegates;

namespace Platform.Data.Doublets.Decorators
{
    public class LoggingDecorator<TLinkAddress> : LinksDecoratorBase<TLinkAddress>
    {
        private readonly Stream _logStream;
        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) =>
            {
                handlerState.Handle(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) =>
            {
                handlerState.Handle(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) =>
            {
                handlerState.Handle(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/NoExceptionsDecorator.cs

```
using System;
using System.Collections.Generic;
using Platform.Delegates;
using Platform.Exceptions;

namespace Platform.Data.Doublets.Decorators;

public class NoExceptionsDecorator<TLinkAddress> : LinksDecoratorBase<TLinkAddress>
{
    public NoExceptionsDecorator(ILinks<TLinkAddress> storage) : base(storage) { }

    public override TLinkAddress Count(IList<TLinkAddress>? restriction)
    {
        try
        {
            return base.Count(restriction);
        }
        catch (Exception exception)
        {
            exception.Ignore();
            return Constants.Error;
        }
    }

    public override TLinkAddress Each(IList<TLinkAddress>? restriction,
    ↪  ReadHandler<TLinkAddress>? handler)
    {
```

```
27          try
28          {
29              return base.Each(restriction, handler);
30          }
31          catch (Exception exception)
32          {
33              exception.Ignore();
34              return Constants.Error;
35          }
36      }
37
38      public override TLinkAddress Create(IList<TLinkAddress>? substitution,
        ↪  WriteHandler<TLinkAddress>? handler)
39      {
40          try
41          {
42              return base.Create(substitution, handler);
43          }
44          catch (Exception exception)
45          {
46              exception.Ignore();
47              return Constants.Error;
48          }
49      }
50
51      public override TLinkAddress Update(IList<TLinkAddress>? restriction, IList<TLinkAddress>?
        ↪  substitution, WriteHandler<TLinkAddress>? handler)
52      {
53          try
54          {
55              return base.Update(restriction, substitution, handler);
56          }
57          catch (Exception exception)
58          {
59              exception.Ignore();
60              return Constants.Error;
61          }
62      }
63
64      public override TLinkAddress Delete(IList<TLinkAddress>? restriction,
        ↪  WriteHandler<TLinkAddress>? handler)
65      {
66          try
67          {
68              return base.Delete(restriction, handler);
69          }
70          catch (Exception exception)
71          {
72              exception.Ignore();
73              return Constants.Error;
74          }
75      }
76  }
```

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

```
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using Platform.Delegates;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Data.Doublets.Decorators
9   {
10      /// <summary>
11      /// <para>
12      /// Represents the non null contents link deletion resolver.
13      /// </para>
14      /// <para></para>
15      /// </summary>
16      /// <seealso cref="LinksDecoratorBase{TLinkAddress}"/>
17      public class NonNullContentsLinkDeletionResolver<TLinkAddress> :
        ↪  LinksDecoratorBase<TLinkAddress>
18      {
19          /// <summary>
20          /// <para>
21          /// Initializes a new <see cref="NonNullContentsLinkDeletionResolver"/> instance.
22          /// </para>
23          /// <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 = _links.GetIndex(restriction);
            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;
        }
    }
}
```

1.16 ./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 = _links.GetIndex(restriction);
            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/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);
```

```
61
62          /// <summary>
63          /// <para>
64          /// Updates the substitution.
65          /// </para>
66          /// <para></para>
67          /// </summary>
68          /// <param name="restriction">
69          /// <para>The substitution.</para>
70          /// <para></para>
71          /// </param>
72          /// <param name="substitution">
73          /// <para>The substitution.</para>
74          /// <para></para>
75          /// </param>
76          /// <returns>
77          /// <para>The TLinkAddress</para>
78          /// <para></para>
79          /// </returns>
80          [MethodImpl(MethodImplOptions.AggressiveInlining)]
81          public override TLinkAddress Update(IList<TLinkAddress>? restriction,
            ↪ IList<TLinkAddress>? substitution, WriteHandler<TLinkAddress>? handler)
82          {
83              var constants = _constants;
84              var indexPartConstant = constants.IndexPart;
85              var sourcePartConstant = constants.SourcePart;
86              var targetPartConstant = constants.TargetPart;
87              var nullConstant = constants.Null;
88              var itselfConstant = constants.Itself;
89              var existedLink = nullConstant;
90              var updatedLink = restriction[indexPartConstant];
91              var newSource = substitution[sourcePartConstant];
92              var newTarget = substitution[targetPartConstant];
93              var links = _links;
94              if (newSource != itselfConstant && newTarget != itselfConstant)
95              {
96                  existedLink = links.SearchOrDefault(newSource, newTarget);
97              }
98              if (existedLink == nullConstant)
99              {
100                 var before = links.GetLink(updatedLink);
101                 if (before[sourcePartConstant] != newSource || before[targetPartConstant] !=
                    ↪ newTarget)
102                 {
103                     var source = newSource == itselfConstant ? updatedLink : newSource;
104                     var target = newTarget == itselfConstant ? updatedLink : newTarget;
105                     return links.Update(new Link<TLinkAddress>(updatedLink, source, target),
                        ↪ handler);
106                 }
107                 return _links.Constants.Continue;
108             }
109             else
110             {
111                 return _facade.MergeAndDelete(updatedLink, existedLink, handler);
112             }
113         }
114
115         /// <summary>
116         /// <para>
117         /// Deletes the substitution.
118         /// </para>
119         /// <para></para>
120         /// </summary>
121         /// <param name="restriction">
122         /// <para>The substitution.</para>
123         /// <para></para>
124         /// </param>
125         [MethodImpl(MethodImplOptions.AggressiveInlining)]
126         public override TLinkAddress Delete(IList<TLinkAddress>? restriction,
            ↪ WriteHandler<TLinkAddress>? handler)
127         {
128             var linkIndex = _links.GetIndex(restriction);
129             var constants = _links.Constants;
130             WriteHandlerState<TLinkAddress> handlerState = new(constants.Continue,
                ↪ constants.Break, handler);
131             handlerState.Apply(_links.EnforceResetValues(linkIndex, handlerState.Handler));
132             handlerState.Apply(_facade.DeleteAllUsages(linkIndex, handlerState.Handler));
133             handlerState.Apply(_links.Delete(restriction, handlerState.Handler));
```

```
            return handlerState.Result;
        }
    }
}
```

1.18 ./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;
```

```
69              }
70          }
71
72          //public static readonly TLinkAddress NullConstant =
            ↪  Use<LinksConstants<TLinkAddress>>.Single.Null;
73          //public static readonly IReadOnlyList<TLinkAddress> NullLink = new
            ↪  ReadOnlyCollection<TLinkAddress>(new List<TLinkAddress> { NullConstant,
            ↪  NullConstant, NullConstant });
74
75          // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution
            ↪  (Links-Expression)
76          /// <summary>
77          /// <para>
78          /// Triggers the restriction.
79          /// </para>
80          /// <para></para>
81          /// </summary>
82          /// <param name="restriction">
83          /// <para>The restriction.</para>
84          /// <para></para>
85          /// </param>
86          /// <param name="matchedHandler">
87          /// <para>The matched handler.</para>
88          /// <para></para>
89          /// </param>
90          /// <param name="substitution">
91          /// <para>The substitution.</para>
92          /// <para></para>
93          /// </param>
94          /// <param name="substitutedHandler">
95          /// <para>The substituted handler.</para>
96          /// <para></para>
97          /// </param>
98          /// <returns>
99          /// <para>The link</para>
100         /// <para></para>
101         /// </returns>
102         public TLinkAddress Trigger(IList<TLinkAddress>? restriction,
            ↪  WriteHandler<TLinkAddress>? matchedHandler, IList<TLinkAddress>? substitution,
            ↪  WriteHandler<TLinkAddress>? substitutedHandler)
103         {
104             ////List<Transition> transitions = null;
105             ////if (!restriction.IsNullOrEmpty())
106             ////{
107             ////    // Есть причина делать проход (чтение)
108             ////    if (matchedHandler != null)
109             ////    {
110             ////        if (!substitution.IsNullOrEmpty())
111             ////        {
112             ////            // restriction => { 0, 0, 0 } | { 0 } // Create
113             ////            // substitution => { itself, 0, 0 } | { itself, itself, itself } //
                ↪  Create / Update
114             ////            // substitution => { 0, 0, 0 } | { 0 } // Delete
115             ////            transitions = new List<Transition>();
116             ////            if (Equals(substitution[Constants.IndexPart], Constants.Null))
117             ////            {
118             ////                // If index is Null, that means we always ignore every other
                ↪  value (they are also Null by definition)
119             ////                var matchDecision = matchedHandler(, NullLink);
120             ////                if (Equals(matchDecision, Constants.Break))
121             ////                    return false;
122             ////                if (!Equals(matchDecision, Constants.Skip))
123             ////                    transitions.Add(new Transition(matchedLink, newValue));
124             ////            }
125             ////            else
126             ////            {
127             ////                Func<T, bool> handler;
128             ////                handler = link =>
129             ////                {
130             ////                    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];
```

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

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

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

        /// <summary>
        /// <para>
        /// Triggers the pattern or condition.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="patternOrCondition">
        /// <para>The pattern or condition.</para>
        /// <para></para>
        /// </param>
        /// <param name="matchHandler">
        /// <para>The match handler.</para>
        /// <para></para>
        /// </param>
        /// <param name="substitution">
        /// <para>The substitution.</para>
        /// <para></para>
        /// </param>
        /// <param name="substitutionHandler">
        /// <para>The substitution handler.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="NotImplementedException">
        /// <para></para>
        /// <para></para>
        /// </exception>
        /// <exception cref="NotSupportedException">
        /// <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))
348                     {
349                         return constants.Break;
350                     }
351                     var after = Array.Empty<TLinkAddress>();
352                     _links.Update(linkToDelete, constants.Null, constants.Null);
353                     _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;
            });
            return changes;
        }
        private TLinkAddress AlwaysContinue(IList<TLinkAddress>? linkToMatch) =>
        ↪ _constants.Continue;
    }
}
```

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

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

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

namespace Platform.Data.Doublets
{

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

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

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

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

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

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

```
78          /// <para>.</para>
79          /// </summury>
80          /// <typeparam>
81          /// <para>.</para>
82          /// <para>.</para>
83          /// </typeparam>
84          /// <param name="other">
85          /// <para>.</para>
86          /// <para>.</para>
87          /// </param>
88          /// <returns>
89          /// <para>.</para>
90          /// <para>.</para>
91          /// </returns>
92          [MethodImpl(MethodImplOptions.AggressiveInlining)]
93          public bool Equals(Doublet<T> other) => _equalityComparer.Equals(Source, other.Source)
            ↪  && _equalityComparer.Equals(Target, other.Target);
94
95          /// <summury>
96          /// <para>.</para>
97          /// <para>.</para>
98          /// </summury>
99          /// <typeparam>
100         /// <para>.</para>
101         /// <para>.</para>
102         /// </typeparam>
103         /// <param name="obj">
104         /// <para>.</para>
105         /// <para>.</para>
106         /// </param>
107         /// <returns>
108         /// <para>.</para>
109         /// <para>.</para>
110         /// </returns>
111         [MethodImpl(MethodImplOptions.AggressiveInlining)]
112         public override bool Equals(object obj) => obj is Doublet<T> doublet ?
            ↪  base.Equals(doublet) : false;
113
114         /// <summury>
115         /// <para>.</para>
116         /// <para>.</para>
117         /// </summury>
118         /// <returns>
119         /// <para>.</para>
120         /// <para>.</para>
121         /// </returns>
122         [MethodImpl(MethodImplOptions.AggressiveInlining)]
123         public override int GetHashCode() => (Source, Target).GetHashCode();
124
125         /// <summury>
126         /// <para>.</para>
127         /// <para>.</para>
128         /// </summury>
129         /// <param name="left">
130         /// <para>.</para>
131         /// <para>.</para>
132         /// </param>
133         /// <param name="right">
134         /// <para>.</para>
135         /// <para>.</para>
136         /// </param>
137         /// <returns>
138         /// <para>.</para>
139         /// <para>.</para>
140         /// </returns>
141         [MethodImpl(MethodImplOptions.AggressiveInlining)]
142         public static bool operator ==(Doublet<T> left, Doublet<T> right) => left.Equals(right);
143
144         /// <summury>
145         /// <para>.</para>
146         /// <para>.</para>
147         /// </summury>
148         /// <param name="left">
149         /// <para>.</para>
150         /// <para>.</para>
151         /// </param>
152         /// <param name="right">
153         /// <para>.</para>
```

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

1.20 ./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();
    }
}
```

1.21 ./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 != null ? handler(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 != null ? handler(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 != null ?
                ↪  handler(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 != null ?
                ↪  handler(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);
            }
        }
    }
}
```

1.22 ./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;
    }

    struct FfiLink_UInt16
    {
        public UInt16 Index;
        public UInt16 Source;
        public UInt16 Target;
    }

```

```
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>
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];
```

```
                    }
                    return from_u8.Convert(Methods.ByteUnitedMemoryLinks_Count(_ptr,
                    ↪  restrictionArray, (nuint)restrictionLength));
                }
                case ushort:
                {
                    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_Count(_ptr,
                    ↪  restrictionArray, (nuint)restrictionLength));
                }
                case uint:
                {
                    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_Count(_ptr,
                    ↪  restrictionArray, (nuint)restrictionLength));
                }
                case ulong:
                {
                    {
                        var restrictionArray = stackalloc ulong[restrictionLength];
                        var ulongRestrictionArray = (IList<ulong>)restriction;
                        for (var i = 0; i < restrictionLength; i++)
                        {
                            restrictionArray[i] = ulongRestrictionArray[i];
                        }
                        return from_u64.Convert(Methods.UInt64UnitedMemoryLinks_Count(_ptr,
                        ↪  restrictionArray, (nuint)restrictionLength));
                    }
                }
                default:
                {
                    throw new NotImplementedException();
                }
            }
        }
    }

    public TLinkAddress Each(IList<TLinkAddress>? restriction, ReadHandler<TLinkAddress>?
    ↪  handler)
    {
        var restrictionLength = restriction?.Count ?? 0;
        unsafe
        {
            TLinkAddress t = default;
            switch (t)
            {
                case byte:
                {
                    byte Callback(FfiLink_UInt8 link) => (byte)from_t.Convert(handler !=
                    ↪  null ? handler(new Link<TLinkAddress>(from_u8.Convert(link.Index),
                    ↪  from_u8.Convert(link.Source), from_u8.Convert(link.Target))) :
                    ↪  Constants.Continue);
                    var restrictionArray = stackalloc byte[restrictionLength];
                    var byteRestrictionArray = (IList<byte>)restriction;
                    for (var i = 0; i < restrictionLength; i++)
                    {
                        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
                    ↪  != null ? handler(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 !=
                    ↪  null ? handler(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 !=
                        ↪  null ? handler(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:
                {
                    byte Callback(FfiLink_UInt8 before, FfiLink_UInt8 after) =>
                        (byte)from_t.Convert(handler != null ? handler(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);
                    var substitutionArray = stackalloc byte[substitutionLength];
                    var byteSubstitutionArray = (IList<byte>)substitution;
                    for (var i = 0; i < substitutionLength; i++)
                    {
                        substitutionArray[i] = byteSubstitutionArray[i];
                    }
                    return from_u8.Convert(Methods.ByteUnitedMemoryLinks_Create(_ptr,
                    ↪  substitutionArray, (nuint)(substitution?.Count ?? 0), Callback));
                }
                case ushort:
```

```
347                     {
348                         ushort Callback(FfiLink_UInt16 before, FfiLink_UInt16 after) =>
                            ↪ (ushort)from_t.Convert(handler != null ? handler(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 != null ? handler(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 != null ? handler(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 != null ? handler(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 != null ? handler(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 != null ? handler(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];
```

```
                        }
                        var substitutionArray = stackalloc ulong[substitutionLength];
                        var ulongSubstitutionArray = (IList<ulong>)substitution;
                        for (var i = 0; i < substitutionLength; i++)
                        {
                            substitutionArray[i] = ulongSubstitutionArray[i];
                        }
                        ulong Callback(FfiLink_UInt64 before, FfiLink_UInt64 after) =>
                            (ulong)from_t.Convert(handler != null ? handler(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);
                        return from_u64.Convert(Methods.UInt64UnitedMemoryLinks_Update(_ptr,
                            ↪ restrictionArray, (nuint)restrictionLength, substitutionArray,
                            ↪ (nuint)(substitution?.Count ?? 0), Callback));
                    }
                    default:
                    {
                        throw new NotImplementedException();
                    }
                }
            }
        }

        public TLinkAddress Delete(IList<TLinkAddress>? restriction, WriteHandler<TLinkAddress>?
        ↪ handler)
        {
            var restrictionLength = restriction?.Count ?? 0;
            unsafe
            {
                TLinkAddress t = default;
                switch (t)
                {
                    case byte:
                    {
                        var restrictionArray = stackalloc byte[restrictionLength];
                        var byteRestrictionArray = (IList<byte>)restriction;
                        for (var i = 0; i < restrictionLength; i++)
                        {
                            restrictionArray[i] = byteRestrictionArray[i];
                        }
                        byte Callback(FfiLink_UInt8 before, FfiLink_UInt8 after) =>
                            (byte)from_t.Convert(handler != null ? handler(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);
                        return (TLinkAddress)(object)Methods.ByteUnitedMemoryLinks_Delete(_ptr,
                            ↪ restrictionArray, (nuint)restrictionLength, Callback);
                    }
                    case ushort:
                    {
                        var restrictionArray = stackalloc ushort[restrictionLength];
                        var ushortRestrictionArray = (IList<ushort>)restriction;
                        for (var i = 0; i < restrictionLength; i++)
                        {
                            restrictionArray[i] = ushortRestrictionArray[i];
                        }
                        ushort Callback(FfiLink_UInt16 before, FfiLink_UInt16 after) =>
                            (ushort)from_t.Convert(handler != null ? handler(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);
                        return
                            ↪ (TLinkAddress)(object)Methods.UInt16UnitedMemoryLinks_Delete(_ptr,
                            ↪ restrictionArray, (nuint)restrictionLength, Callback);
                    }
                    case uint:
                    {
                        var restrictionArray = stackalloc uint[restrictionLength];
                        var uintRestrictionArray = (IList<uint>)restriction;
                        for (var i = 0; i < restrictionLength; i++)
                        {
```

```
                            restrictionArray[i] = uintRestrictionArray[i];
                        }
                        uint Callback(FfiLink_UInt32 before, FfiLink_UInt32 after) =>
                            (uint)from_t.Convert(handler != null ? handler(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);
                        return
                            (TLinkAddress)(object)Methods.UInt32UnitedMemoryLinks_Delete(_ptr,
                            restrictionArray, (nuint)restrictionLength, Callback);
                    }
                    case ulong:
                    {
                        var restrictionArray = stackalloc ulong[restrictionLength];
                        var ulongRestrictionArray = (IList<ulong>)restriction;
                        for (var i = 0; i < restrictionLength; i++)
                        {
                            restrictionArray[i] = ulongRestrictionArray[i];
                        }
                        ulong Callback(FfiLink_UInt64 before, FfiLink_UInt64 after) =>
                            (ulong)from_t.Convert(handler != null ? handler(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);
                        return
                            (TLinkAddress)(object)Methods.UInt64UnitedMemoryLinks_Delete(_ptr,
                            restrictionArray, (nuint)restrictionLength, Callback);
                    }
                    default:
                    {
                        throw new NotImplementedException();
                    }
                }
            }
        }

        protected override void Dispose(bool manual, bool wasDisposed)
        {
            unsafe
            {
                if (wasDisposed && _ptr != null)
                {
                    return;
                }
                TLinkAddress t = default;
                switch (t)
                    {
                        case byte:
                            Methods.ByteUnitedMemoryLinks_Drop(_ptr);
                            break;
                        case ushort:
                            Methods.UInt16UnitedMemoryLinks_Drop(_ptr);
                            break;
                        case uint:
                            Methods.UInt32UnitedMemoryLinks_Drop(_ptr);
                            break;
                        case ulong:
                            Methods.UInt64UnitedMemoryLinks_Drop(_ptr);
                            break;
                        default:
                            throw new NotImplementedException();
                    }
            }
        }
    }
}
```

1.23 ./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>>
    {
    }
}
```

1.24 ./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 ? 0UL : linksCount - amountOfDeletions;
            for (var i = 0UL; i < amountOfDeletions; i++)
            {
                linksCount = addressToUInt64Converter.Convert(links.Count());
                if (linksCount <= min)
                {
                    break;
                }
                var linksAddressRange = new Range<ulong>(min, linksCount);
                var link =
                ↪  uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
                links.Delete(link);
            }
        }

        /// <summary>
        /// <para>
        /// Deletes the links.
```

```
136         /// </para>
137         /// <para></para>
138         /// </summary>
139         /// <typeparam name="TLinkAddress">
140         /// <para>The link.</para>
141         /// <para></para>
142         /// </typeparam>
143         /// <param name="links">
144         /// <para>The links.</para>
145         /// <para></para>
146         /// </param>
147         /// <param name="linkToDelete">
148         /// <para>The link to delete.</para>
149         /// <para></para>
150         /// </param>
151         [MethodImpl(MethodImplOptions.AggressiveInlining)]
152         public static TLinkAddress Delete<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪ TLinkAddress linkToDelete, WriteHandler<TLinkAddress>? handler)
153         {
154             if (links.Exists(linkToDelete))
155             {
156                 links.EnforceResetValues(linkToDelete, handler);
157             }
158             return links.Delete(new LinkAddress<TLinkAddress>(linkToDelete), handler);
159         }
160
161         /// <remarks>
162         /// TODO: Возможно есть очень простой способ это сделать.
163         /// (Например просто удалить файл, или изменить его размер таким образом,
164         /// чтобы удалился весь контент)
165         /// Например через _header->AllocatedLinks в ResizableDirectMemoryLinks
166         /// </remarks>
167         [MethodImpl(MethodImplOptions.AggressiveInlining)]
168         public static void DeleteAll<TLinkAddress>(this ILinks<TLinkAddress> links)
169         {
170             var equalityComparer = EqualityComparer<TLinkAddress>.Default;
171             var comparer = Comparer<TLinkAddress>.Default;
172             for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
                ↪ Arithmetic.Decrement(i))
173             {
174                 links.Delete(i);
175                 if (!equalityComparer.Equals(links.Count(), Arithmetic.Decrement(i)))
176                 {
177                     i = links.Count();
178                 }
179             }
180         }
181
182         /// <summary>
183         /// <para>
184         /// Firsts the links.
185         /// </para>
186         /// <para></para>
187         /// </summary>
188         /// <typeparam name="TLinkAddress">
189         /// <para>The link.</para>
190         /// <para></para>
191         /// </typeparam>
192         /// <param name="links">
193         /// <para>The links.</para>
194         /// <para></para>
195         /// </param>
196         /// <exception cref="InvalidOperationException">
197         /// <para>В процессе поиска по хранилищу не было найдено связей.</para>
198         /// <para></para>
199         /// </exception>
200         /// <exception cref="InvalidOperationException">
201         /// <para>В хранилище нет связей.</para>
202         /// <para></para>
203         /// </exception>
204         /// <returns>
205         /// <para>The first link.</para>
206         /// <para></para>
207         /// </returns>
208         [MethodImpl(MethodImplOptions.AggressiveInlining)]
209         public static TLinkAddress First<TLinkAddress>(this ILinks<TLinkAddress> links)
210         {
211             TLinkAddress firstLink = default;
```

```
212             var equalityComparer = EqualityComparer<TLinkAddress>.Default;
213             if (equalityComparer.Equals(links.Count(), default))
214             {
215                 throw new InvalidOperationException("В хранилище нет связей.");
216             }
217             links.Each(new Link<TLinkAddress>(links.Constants.Any, links.Constants.Any,
                ↪  links.Constants.Any), link =>
218             {
219                 firstLink = link[links.Constants.IndexPart];
220                 return links.Constants.Break;
221             });
222             if (equalityComparer.Equals(firstLink, default))
223             {
224                 throw new InvalidOperationException("В процессе поиска по хранилищу не было
                    ↪  найдено связей.");
225             }
226             return firstLink;
227         }
228 
229         /// <summary>
230         /// <para>
231         /// Singles the or default using the specified links.
232         /// </para>
233         /// <para></para>
234         /// </summary>
235         /// <typeparam name="TLinkAddress">
236         /// <para>The link.</para>
237         /// <para></para>
238         /// </typeparam>
239         /// <param name="links">
240         /// <para>The links.</para>
241         /// <para></para>
242         /// </param>
243         /// <param name="query">
244         /// <para>The query.</para>
245         /// <para></para>
246         /// </param>
247         /// <returns>
248         /// <para>The result.</para>
249         /// <para></para>
250         /// </returns>
251         [MethodImpl(MethodImplOptions.AggressiveInlining)]
252         public static IList<TLinkAddress>? SingleOrDefault<TLinkAddress>(this
            ↪  ILinks<TLinkAddress> links, IList<TLinkAddress>? query)
253         {
254             IList<TLinkAddress>? result = null;
255             var count = 0;
256             var constants = links.Constants;
257             var @continue = constants.Continue;
258             var @break = constants.Break;
259             links.Each(query, linkHandler);
260             return result;
261 
262             TLinkAddress linkHandler(IList<TLinkAddress>? link)
263             {
264                 if (count == 0)
265                 {
266                     result = link;
267                     count++;
268                     return @continue;
269                 }
270                 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 = links.GetSource(values);
303                 var target = links.GetTarget(values);
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>
```

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

        #endregion

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

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

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

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

        /// <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         /// <para>
438         /// Alls the links.
439         /// </para>
440         /// <para></para>
441         /// </summary>
442         /// <typeparam name="TLinkAddress">
443         /// <para>The link.</para>
444         /// <para></para>
445         /// </typeparam>
446         /// <param name="links">
447         /// <para>The links.</para>
448         /// <para></para>
449         /// </param>
450         /// <param name="restriction">
451         /// <para>The restriction.</para>
452         /// <para></para>
453         /// </param>
454         /// <returns>
455         /// <para>A list of i list t link</para>
456         /// <para></para>
457         /// </returns>
458         [MethodImpl(MethodImplOptions.AggressiveInlining)]
459         public static IList<IList<TLinkAddress>?> All<TLinkAddress>(this ILinks<TLinkAddress>
            ↪ links, params TLinkAddress[] restriction)
460         {
461             var allLinks = new List<IList<TLinkAddress>?>();
462             var filler = new ListFiller<IList<TLinkAddress>?, TLinkAddress>(allLinks,
                ↪ links.Constants.Continue);
463             links.Each(filler.AddAndReturnConstant, restriction);
464             return allLinks;
465         }
466
467         /// <summary>
468         /// <para>
469         /// Alls the indices using the specified links.
470         /// </para>
471         /// <para></para>
472         /// </summary>
473         /// <typeparam name="TLinkAddress">
474         /// <para>The link.</para>
475         /// <para></para>
476         /// </typeparam>
477         /// <param name="links">
478         /// <para>The links.</para>
479         /// <para></para>
480         /// </param>
481         /// <param name="restriction">
482         /// <para>The restriction.</para>
483         /// <para></para>
484         /// </param>
485         /// <returns>
486         /// <para>A list of t link</para>
487         /// <para></para>
488         /// </returns>
489         [MethodImpl(MethodImplOptions.AggressiveInlining)]
490         public static IList<TLinkAddress>? AllIndices<TLinkAddress>(this ILinks<TLinkAddress>
            ↪ links, params TLinkAddress[] restriction)
491         {
492             var allIndices = new List<TLinkAddress>();
493             var filler = new ListFiller<TLinkAddress, TLinkAddress>(allIndices,
                ↪ links.Constants.Continue);
494             links.Each(filler.AddFirstAndReturnConstant, restriction);
495             return allIndices;
496         }
497
498         /// <summary>
```

```
499         /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
            ↪ в хранилище связей.
500         /// </summary>
501         /// <param name="links">Хранилище связей.</param>
502         /// <param name="source">Начало связи.</param>
503         /// <param name="target">Конец связи.</param>
504         /// <returns>Значение, определяющее существует ли связь.</returns>
505         [MethodImpl(MethodImplOptions.AggressiveInlining)]
506         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;
507
508         #region Ensure
509         // TODO: May be move to EnsureExtensions or make it both there and here
510
511         /// <summary>
512         /// <para>
513         /// Ensures the link exists using the specified links.
514         /// </para>
515         /// <para></para>
516         /// </summary>
517         /// <typeparam name="TLinkAddress">
518         /// <para>The link.</para>
519         /// <para></para>
520         /// </typeparam>
521         /// <param name="links">
522         /// <para>The links.</para>
523         /// <para></para>
524         /// </param>
525         /// <param name="restriction">
526         /// <para>The restriction.</para>
527         /// <para></para>
528         /// </param>
529         /// <exception cref="ArgumentLinkDoesNotExistsException{TLinkAddress}">
530         /// <para>sequence[{i}]</para>
531         /// <para></para>
532         /// </exception>
533         [MethodImpl(MethodImplOptions.AggressiveInlining)]
534         public static void EnsureLinkExists<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪ IList<TLinkAddress>? restriction)
535         {
536             for (var i = 0; i < restriction.Count; i++)
537             {
538                 if (!links.Exists(restriction[i]))
539                 {
540                     throw new ArgumentLinkDoesNotExistsException<TLinkAddress>(restriction[i],
                        ↪ $"sequence[{i}]");
541                 }
542             }
543         }
544
545         /// <summary>
546         /// <para>
547         /// Ensures the inner reference exists using the specified links.
548         /// </para>
549         /// <para></para>
550         /// </summary>
551         /// <typeparam name="TLinkAddress">
552         /// <para>The link.</para>
553         /// <para></para>
554         /// </typeparam>
555         /// <param name="links">
556         /// <para>The links.</para>
557         /// <para></para>
558         /// </param>
559         /// <param name="reference">
560         /// <para>The reference.</para>
561         /// <para></para>
562         /// </param>
563         /// <param name="argumentName">
564         /// <para>The argument name.</para>
565         /// <para></para>
566         /// </param>
567         /// <exception cref="ArgumentLinkDoesNotExistsException{TLinkAddress}">
568         /// <para></para>
569         /// <para></para>
```

```
570         /// </exception>
571         [MethodImpl(MethodImplOptions.AggressiveInlining)]
572         public static void EnsureInnerReferenceExists<TLinkAddress>(this ILinks<TLinkAddress>
            ↪ links, TLinkAddress reference, string argumentName)
573         {
574             if (links.Constants.IsInternalReference(reference) && !links.Exists(reference))
575             {
576                 throw new ArgumentLinkDoesNotExistsException<TLinkAddress>(reference,
                    ↪ argumentName);
577             }
578         }
579 
580         /// <summary>
581         /// <para>
582         /// Ensures the inner reference exists using the specified links.
583         /// </para>
584         /// <para></para>
585         /// </summary>
586         /// <typeparam name="TLinkAddress">
587         /// <para>The link.</para>
588         /// <para></para>
589         /// </typeparam>
590         /// <param name="links">
591         /// <para>The links.</para>
592         /// <para></para>
593         /// </param>
594         /// <param name="restriction">
595         /// <para>The restriction.</para>
596         /// <para></para>
597         /// </param>
598         /// <param name="argumentName">
599         /// <para>The argument name.</para>
600         /// <para></para>
601         /// </param>
602         [MethodImpl(MethodImplOptions.AggressiveInlining)]
603         public static void EnsureInnerReferenceExists<TLinkAddress>(this ILinks<TLinkAddress>
            ↪ links, IList<TLinkAddress>? restriction, string argumentName)
604         {
605             for (int i = 0; i < restriction.Count; i++)
606             {
607                 links.EnsureInnerReferenceExists(restriction[i], argumentName);
608             }
609         }
610 
611         /// <summary>
612         /// <para>
613         /// Ensures the link is any or exists using the specified links.
614         /// </para>
615         /// <para></para>
616         /// </summary>
617         /// <typeparam name="TLinkAddress">
618         /// <para>The link.</para>
619         /// <para></para>
620         /// </typeparam>
621         /// <param name="links">
622         /// <para>The links.</para>
623         /// <para></para>
624         /// </param>
625         /// <param name="restriction">
626         /// <para>The restriction.</para>
627         /// <para></para>
628         /// </param>
629         /// <exception cref="ArgumentLinkDoesNotExistsException{TLinkAddress}">
630         /// <para>sequence[{i}]</para>
631         /// <para></para>
632         /// </exception>
633         [MethodImpl(MethodImplOptions.AggressiveInlining)]
634         public static void EnsureLinkIsAnyOrExists<TLinkAddress>(this ILinks<TLinkAddress>
            ↪ links, IList<TLinkAddress>? restriction)
635         {
636             var equalityComparer = EqualityComparer<TLinkAddress>.Default;
637             var any = links.Constants.Any;
638             for (var i = 0; i < restriction.Count; i++)
639             {
640                 if (!equalityComparer.Equals(restriction[i], any) &&
                    ↪ !links.Exists(restriction[i]))
641                 {
```

```
642                     throw new ArgumentLinkDoesNotExistsException<TLinkAddress>(restriction[i],
                        ↪ $"sequence[{i}]");
643                 }
644             }
645         }
646 
647         /// <summary>
648         /// <para>
649         /// Ensures the link is any or exists using the specified links.
650         /// </para>
651         /// <para></para>
652         /// </summary>
653         /// <typeparam name="TLinkAddress">
654         /// <para>The link.</para>
655         /// <para></para>
656         /// </typeparam>
657         /// <param name="links">
658         /// <para>The links.</para>
659         /// <para></para>
660         /// </param>
661         /// <param name="link">
662         /// <para>The link.</para>
663         /// <para></para>
664         /// </param>
665         /// <param name="argumentName">
666         /// <para>The argument name.</para>
667         /// <para></para>
668         /// </param>
669         /// <exception cref="ArgumentLinkDoesNotExistsException{TLinkAddress}">
670         /// <para></para>
671         /// <para></para>
672         /// </exception>
673         [MethodImpl(MethodImplOptions.AggressiveInlining)]
674         public static void EnsureLinkIsAnyOrExists<TLinkAddress>(this ILinks<TLinkAddress>
            ↪ links, TLinkAddress link, string argumentName)
675         {
676             var equalityComparer = EqualityComparer<TLinkAddress>.Default;
677             if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
678             {
679                 throw new ArgumentLinkDoesNotExistsException<TLinkAddress>(link, argumentName);
680             }
681         }
682 
683         /// <summary>
684         /// <para>
685         /// Ensures the link is itself or exists using the specified links.
686         /// </para>
687         /// <para></para>
688         /// </summary>
689         /// <typeparam name="TLinkAddress">
690         /// <para>The link.</para>
691         /// <para></para>
692         /// </typeparam>
693         /// <param name="links">
694         /// <para>The links.</para>
695         /// <para></para>
696         /// </param>
697         /// <param name="link">
698         /// <para>The link.</para>
699         /// <para></para>
700         /// </param>
701         /// <param name="argumentName">
702         /// <para>The argument name.</para>
703         /// <para></para>
704         /// </param>
705         /// <exception cref="ArgumentLinkDoesNotExistsException{TLinkAddress}">
706         /// <para></para>
707         /// <para></para>
708         /// </exception>
709         [MethodImpl(MethodImplOptions.AggressiveInlining)]
710         public static void EnsureLinkIsItselfOrExists<TLinkAddress>(this ILinks<TLinkAddress>
            ↪ links, TLinkAddress link, string argumentName)
711         {
712             var equalityComparer = EqualityComparer<TLinkAddress>.Default;
713             if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
714             {
715                 throw new ArgumentLinkDoesNotExistsException<TLinkAddress>(link, argumentName);
716             }
```

```
717         }
718
719         /// <param name="links">Хранилище связей.</param>
720         [MethodImpl(MethodImplOptions.AggressiveInlining)]
721         public static void EnsureDoesNotExists<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪ TLinkAddress source, TLinkAddress target)
722         {
723             if (links.Exists(source, target))
724             {
725                 throw new LinkWithSameValueAlreadyExistsException();
726             }
727         }
728
729         /// <param name="links">Хранилище связей.</param>
730         [MethodImpl(MethodImplOptions.AggressiveInlining)]
731         public static void EnsureNoUsages<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪ TLinkAddress link)
732         {
733             if (links.HasUsages(link))
734             {
735                 throw new ArgumentLinkHasDependenciesException<TLinkAddress>(link);
736             }
737         }
738
739         /// <param name="links">Хранилище связей.</param>
740         [MethodImpl(MethodImplOptions.AggressiveInlining)]
741         public static void EnsureCreated<TLinkAddress>(this ILinks<TLinkAddress> links, params
            ↪ TLinkAddress[] addresses)  => links.EnsureCreated(links.Create, addresses);
742
743         /// <param name="links">Хранилище связей.</param>
744         [MethodImpl(MethodImplOptions.AggressiveInlining)]
745         public static void EnsurePointsCreated<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪ params TLinkAddress[] addresses)  => links.EnsureCreated(links.CreatePoint,
            ↪ addresses);
746
747         /// <param name="links">Хранилище связей.</param>
748         [MethodImpl(MethodImplOptions.AggressiveInlining)]
749         public static void EnsureCreated<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪ Func<TLinkAddress> creator, params TLinkAddress[] addresses)
750         {
751             var addressToUInt64Converter = CheckedConverter<TLinkAddress, ulong>.Default;
752             var uInt64ToAddressConverter = CheckedConverter<ulong, TLinkAddress>.Default;
753             var nonExistentAddresses = new HashSet<TLinkAddress>(addresses.Where(x =>
                ↪ !links.Exists(x)));
754             if (nonExistentAddresses.Count > 0)
755             {
756                 var max = nonExistentAddresses.Max();
757                 max = uInt64ToAddressConverter.Convert(System.Math.Min(addressToUInt64Converter.
                    ↪ Convert(max),
                    ↪ addressToUInt64Converter.Convert(links.Constants.InternalReferencesRange.Max
                    ↪ imum)));
758                 var createdLinks = new List<TLinkAddress>();
759                 var equalityComparer = EqualityComparer<TLinkAddress>.Default;
760                 TLinkAddress createdLink = creator();
761                 while (!equalityComparer.Equals(createdLink, max))
762                 {
763                     createdLinks.Add(createdLink);
764                 }
765                 for (var i = 0; i < createdLinks.Count; i++)
766                 {
767                     if (!nonExistentAddresses.Contains(createdLinks[i]))
768                     {
769                         links.Delete(createdLinks[i]);
770                     }
771                 }
772             }
773         }
774
775         #endregion
776
777         /// <param name="links">Хранилище связей.</param>
778         [MethodImpl(MethodImplOptions.AggressiveInlining)]
779         public static TLinkAddress CountUsages<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪ TLinkAddress link)
780         {
781             var constants = links.Constants;
782             var values = links.GetLink(link);
```

```
783             TLinkAddress usagesAsSource = links.Count(new Link<TLinkAddress>(constants.Any,
                ↪  link, constants.Any));
784             var equalityComparer = EqualityComparer<TLinkAddress>.Default;
785             if (equalityComparer.Equals(links.GetSource(values), link))
786             {
787                 usagesAsSource = Arithmetic<TLinkAddress>.Decrement(usagesAsSource);
788             }
789             TLinkAddress usagesAsTarget = links.Count(new Link<TLinkAddress>(constants.Any,
                ↪  constants.Any, link));
790             if (equalityComparer.Equals(links.GetTarget(values), link))
791             {
792                 usagesAsTarget = Arithmetic<TLinkAddress>.Decrement(usagesAsTarget);
793             }
794             return Arithmetic<TLinkAddress>.Add(usagesAsSource, usagesAsTarget);
795         }
796
797         /// <param name="links">Хранилище связей.</param>
798         [MethodImpl(MethodImplOptions.AggressiveInlining)]
799         public static bool HasUsages<TLinkAddress>(this ILinks<TLinkAddress> links, TLinkAddress
                ↪  link)  => Comparer<TLinkAddress>.Default.Compare(links.CountUsages(link), default) >
                ↪  0;
800
801         /// <param name="links">Хранилище связей.</param>
802         [MethodImpl(MethodImplOptions.AggressiveInlining)]
803         public static bool Equals<TLinkAddress>(this ILinks<TLinkAddress> links, TLinkAddress
                ↪  link, TLinkAddress source, TLinkAddress target)
804         {
805             var constants = links.Constants;
806             var values = links.GetLink(link);
807             var equalityComparer = EqualityComparer<TLinkAddress>.Default;
808             return equalityComparer.Equals(links.GetSource(values), source) &&
                    ↪  equalityComparer.Equals(links.GetTarget(values), target);
809         }
810
811         /// <summary>
812         /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
813         /// </summary>
814         /// <param name="links">Хранилище связей.</param>
815         /// <param name="source">Индекс связи, которая является началом для искомой
                ↪  связи.</param>
816         /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
817         /// <returns>Индекс искомой связи с указанными Source (началом) и Target
                ↪  (концом).</returns>
818         [MethodImpl(MethodImplOptions.AggressiveInlining)]
819         public static TLinkAddress SearchOrDefault<TLinkAddress>(this ILinks<TLinkAddress>
                ↪  links, TLinkAddress source, TLinkAddress target)
820         {
821             var contants = links.Constants;
822             var setter = new Setter<TLinkAddress, TLinkAddress>(contants.Continue,
                    ↪  contants.Break, default);
823             links.Each(setter.SetFirstAndReturnFalse, contants.Any, source, target);
824             return setter.Result;
825         }
826
827         public static TLinkAddress CreatePoint<TLinkAddress>(this ILinks<TLinkAddress> links)
828         {
829             var constants = links.Constants;
830             var setter = new Setter<TLinkAddress, TLinkAddress>(constants.Continue,
                    ↪  constants.Break);
831             links.CreatePoint(setter.SetFirstFromSecondListAndReturnTrue);
832             return setter.Result;
833         }
834
835         /// <param name="links">Хранилище связей.</param>
836         [MethodImpl(MethodImplOptions.AggressiveInlining)]
837         public static TLinkAddress CreatePoint<TLinkAddress>(this ILinks<TLinkAddress> links,
                ↪  WriteHandler<TLinkAddress>? handler)
838         {
839             var constants = links.Constants;
840             WriteHandlerState<TLinkAddress> handlerState = new(constants.Continue,
                    ↪  constants.Break, handler);
841             TLinkAddress link = default;
842             TLinkAddress HandlerWrapper(IList<TLinkAddress>? before, IList<TLinkAddress>? after)
843             {
844                 link = links.GetIndex(after);
845                 return handlerState.Handle(before, after);;
846             }
847             handlerState.Apply(links.Create(null, HandlerWrapper));
```

```
848             handlerState.Apply(links.Update(link, link, link, HandlerWrapper));
849             return handlerState.Result;
850         }
851 
852         public static TLinkAddress CreateAndUpdate<TLinkAddress>(this ILinks<TLinkAddress>
        ↪  links, TLinkAddress source, TLinkAddress target)
853         {
854             var constants = links.Constants;
855             var setter = new Setter<TLinkAddress, TLinkAddress>(constants.Continue,
            ↪  constants.Break);
856             links.CreateAndUpdate(source, target, setter.SetFirstFromSecondListAndReturnTrue);
857             return setter.Result;
858         }
859 
860 
861         /// <param name="links">Хранилище связей.</param>
862         [MethodImpl(MethodImplOptions.AggressiveInlining)]
863         public static TLinkAddress CreateAndUpdate<TLinkAddress>(this ILinks<TLinkAddress>
        ↪  links, TLinkAddress source, TLinkAddress target, WriteHandler<TLinkAddress>? handler)
864         {
865             var constants = links.Constants;
866             TLinkAddress createdLink = default;
867             WriteHandlerState<TLinkAddress> handlerState = new(constants.Continue,
            ↪  constants.Break, handler);
868             handlerState.Apply(links.Create(null, (before, after) =>
869             {
870                 createdLink = links.GetIndex(after);
871                 return handlerState.Handle(before, after);;
872             }));
873             handlerState.Apply(links.Update(createdLink, source, target, handler));
874             return handlerState.Result;
875         }
876 
877         /// <summary>
878         /// Обновляет связь с указанными началом (Source) и концом (Target)
879         /// на связь с указанными началом (NewSource) и концом (NewTarget).
880         /// </summary>
881         /// <param name="links">Хранилище связей.</param>
882         /// <param name="link">Индекс обновляемой связи.</param>
883         /// <param name="newSource">Индекс связи, которая является началом связи, на которую
        ↪  выполняется обновление.</param>
884         /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
        ↪  выполняется обновление.</param>
885         /// <returns>Индекс обновлённой связи.</returns>
886         [MethodImpl(MethodImplOptions.AggressiveInlining)]
887         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));
888 
889         public static TLinkAddress Update<TLinkAddress>(this ILinks<TLinkAddress> links, params
        ↪  TLinkAddress[] restriction)  => links.Update((IList<TLinkAddress>)restriction);
890 
891         public static TLinkAddress Update<TLinkAddress>(this ILinks<TLinkAddress> links,
        ↪  WriteHandler<TLinkAddress>? handler, params TLinkAddress[] restriction)  =>
        ↪  links.Update(restriction, handler);
892 
893         public static TLinkAddress Update<TLinkAddress>(this ILinks<TLinkAddress> links,
        ↪  IList<TLinkAddress>? restriction)
894         {
895             var constants = links.Constants;
896             var setter = new Setter<TLinkAddress, TLinkAddress>(constants.Continue,
            ↪  constants.Break);
897             links.Update(restriction, setter.SetFirstFromSecondListAndReturnTrue);
898             return setter.Result;
899         }
900 
901 
902         /// <summary>
903         /// Обновляет связь с указанными началом (Source) и концом (Target)
904         /// на связь с указанными началом (NewSource) и концом (NewTarget).
905         /// </summary>
906         /// <param name="links">Хранилище связей.</param>
907         /// <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>
908         /// <returns>Индекс обновлённой связи.</returns>
909         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
910         public static TLinkAddress Update<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪ IList<TLinkAddress>? restriction, WriteHandler<TLinkAddress>? handler)
911         {
912             return restriction.Count switch
913             {
914                 2 => links.MergeAndDelete(restriction[0], restriction[1], handler),
915                 4 => links.UpdateOrCreateOrGet(restriction[0], restriction[1], restriction[2],
                    ↪ restriction[3], handler),
916                 _ => links.Update(restriction[0], restriction[1], restriction[2], handler)
917             };
918         }
919 
920         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);
921 
922         /// <summary>
923         /// <para>
924         /// Resolves the constant as self reference using the specified links.
925         /// </para>
926         /// <para></para>
927         /// </summary>
928         /// <typeparam name="TLinkAddress">
929         /// <para>The link.</para>
930         /// <para></para>
931         /// </typeparam>
932         /// <param name="links">
933         /// <para>The links.</para>
934         /// <para></para>
935         /// </param>
936         /// <param name="constant">
937         /// <para>The constant.</para>
938         /// <para></para>
939         /// </param>
940         /// <param name="restriction">
941         /// <para>The restriction.</para>
942         /// <para></para>
943         /// </param>
944         /// <param name="substitution">
945         /// <para>The substitution.</para>
946         /// <para></para>
947         /// </param>
948         /// <returns>
949         /// <para>A list of t link</para>
950         /// <para></para>
951         /// </returns>
952         [MethodImpl(MethodImplOptions.AggressiveInlining)]
953         public static IList<TLinkAddress>? ResolveConstantAsSelfReference<TLinkAddress>(this
            ↪ ILinks<TLinkAddress> links, TLinkAddress constant, IList<TLinkAddress>? restriction,
            ↪ IList<TLinkAddress>? substitution)
954         {
955             var equalityComparer = EqualityComparer<TLinkAddress>.Default;
956             var constants = links.Constants;
957             var restrictionIndex = links.GetIndex(restriction);
958             var substitutionIndex = links.GetIndex(substitution);
959             if (equalityComparer.Equals(substitutionIndex, default))
960             {
961                 substitutionIndex = restrictionIndex;
962             }
963             var source = links.GetSource(substitution);
964             var target = links.GetTarget(substitution);
965             source = equalityComparer.Equals(source, constant) ? substitutionIndex : source;
966             target = equalityComparer.Equals(target, constant) ? substitutionIndex : target;
967             return new Link<TLinkAddress>(substitutionIndex, source, target);
968         }
969 
970         /// <summary>
971         /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
            ↪ с указанными Source (началом) и Target (концом).
972         /// </summary>
973         /// <param name="links">Хранилище связей.</param>
974         /// <param name="source">Индекс связи, которая является началом на создаваемой
            ↪ связи.</param>
975         /// <param name="target">Индекс связи, которая является концом для создаваемой
            ↪ связи.</param>
976         /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
```

```
977         [MethodImpl(MethodImplOptions.AggressiveInlining)]
978         public static TLinkAddress GetOrCreate<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪ TLinkAddress source, TLinkAddress target)
979         {
980             var link = links.SearchOrDefault(source, target);
981             if (EqualityComparer<TLinkAddress>.Default.Equals(link, default))
982             {
983                 link = links.CreateAndUpdate(source, target);
984             }
985             return link;
986         }
987
988         public static TLinkAddress UpdateOrCreateOrGet<TLinkAddress>(this ILinks<TLinkAddress>
            ↪ links, TLinkAddress source, TLinkAddress target, TLinkAddress newSource,
            ↪ TLinkAddress newTarget)
989         {
990             var constants = links.Constants;
991             var setter = new Setter<TLinkAddress, TLinkAddress>(constants.Continue,
                ↪ constants.Break);
992             links.UpdateOrCreateOrGet(source, target, newSource, newTarget,
                ↪ setter.SetFirstFromSecondListAndReturnTrue);
993             return setter.Result;
994         }
995
996         /// <summary>
997         /// Обновляет связь с указанными началом (Source) и концом (Target)
998         /// на связь с указанными началом (NewSource) и концом (NewTarget).
999         /// </summary>
1000        /// <param name="links">Хранилище связей.</param>
1001        /// <param name="source">Индекс связи, которая является началом обновляемой
            ↪ связи.</param>
1002        /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
1003        /// <param name="newSource">Индекс связи, которая является началом связи, на которую
            ↪ выполняется обновление.</param>
1004        /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
            ↪ выполняется обновление.</param>
1005        /// <returns>Индекс обновлённой связи.</returns>
1006        [MethodImpl(MethodImplOptions.AggressiveInlining)]
1007        public static TLinkAddress UpdateOrCreateOrGet<TLinkAddress>(this ILinks<TLinkAddress>
            ↪ links, TLinkAddress source, TLinkAddress target, TLinkAddress newSource,
            ↪ TLinkAddress newTarget, WriteHandler<TLinkAddress>? handler)
1008        {
1009            var equalityComparer = EqualityComparer<TLinkAddress>.Default;
1010            var link = links.SearchOrDefault(source, target);
1011            if (equalityComparer.Equals(link, default))
1012            {
1013                return links.CreateAndUpdate(newSource, newTarget, handler);
1014            }
1015            if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
                ↪ target))
1016            {
1017                var linkStruct = new Link<TLinkAddress>(link, source, target);
1018                return link;
1019            }
1020            return links.Update(link, newSource, newTarget, handler);
1021        }
1022
1023        /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
1024        /// <param name="links">Хранилище связей.</param>
1025        /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
1026        /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
1027        [MethodImpl(MethodImplOptions.AggressiveInlining)]
1028        public static TLinkAddress DeleteIfExists<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪ TLinkAddress source, TLinkAddress target)
1029        {
1030            var link = links.SearchOrDefault(source, target);
1031            if (!EqualityComparer<TLinkAddress>.Default.Equals(link, default))
1032            {
1033                links.Delete(link);
1034                return link;
1035            }
1036            return default;
1037        }
1038
1039        /// <summary>Удаляет несколько связей.</summary>
1040        /// <param name="links">Хранилище связей.</param>
1041        /// <param name="deletedLinks">Список адресов связей к удалению.</param>
1042        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
1043        public static void DeleteMany<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪ IList<TLinkAddress>? deletedLinks)
1044        {
1045            for (int i = 0; i < deletedLinks.Count; i++)
1046            {
1047                links.Delete(deletedLinks[i]);
1048            }
1049        }
1050
1051        public static void DeleteAllUsages<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪ TLinkAddress linkIndex)  => links.DeleteAllUsages(linkIndex, null);
1052
1053        /// <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>
1054        [MethodImpl(MethodImplOptions.AggressiveInlining)]
1055        public static TLinkAddress DeleteAllUsages<TLinkAddress>(this ILinks<TLinkAddress>
            ↪ links, TLinkAddress linkIndex, WriteHandler<TLinkAddress>? handler)
1056        {
1057            var constants = links.Constants;
1058            var any = constants.Any;
1059            var equalityComparer = EqualityComparer<TLinkAddress>.Default;
1060            var usagesAsSourceQuery = new Link<TLinkAddress>(any, linkIndex, any);
1061            var usagesAsTargetQuery = new Link<TLinkAddress>(any, any, linkIndex);
1062            var usages = new List<IList<TLinkAddress>?>();
1063            var usagesFiller = new ListFiller<IList<TLinkAddress>?, TLinkAddress>(usages,
                ↪ constants.Continue);
1064            links.Each(usagesFiller.AddAndReturnConstant, usagesAsSourceQuery);
1065            links.Each(usagesFiller.AddAndReturnConstant, usagesAsTargetQuery);
1066            WriteHandlerState<TLinkAddress> handlerState = new(constants.Continue,
                ↪ constants.Break, handler);
1067            foreach (var usage in usages)
1068            {
1069                if (equalityComparer.Equals(links.GetIndex(usage), linkIndex) ||
                    ↪ !links.Exists(links.GetIndex(usage)))
1070                {
1071                    continue;
1072                }
1073                handlerState.Apply(links.Delete(links.GetIndex(usage), handlerState.Handler));
1074            }
1075            return handlerState.Result;
1076        }
1077
1078        /// <summary>
1079        /// <para>
1080        /// Deletes the by query using the specified links.
1081        /// </para>
1082        /// <para></para>
1083        /// </summary>
1084        /// <typeparam name="TLinkAddress">
1085        /// <para>The link.</para>
1086        /// <para></para>
1087        /// </typeparam>
1088        /// <param name="links">
1089        /// <para>The links.</para>
1090        /// <para></para>
1091        /// </param>
1092        /// <param name="query">
1093        /// <para>The query.</para>
1094        /// <para></para>
1095        /// </param>
1096        [MethodImpl(MethodImplOptions.AggressiveInlining)]
1097        public static void DeleteByQuery<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪ Link<TLinkAddress> query)
1098        {
1099            var queryResult = new List<TLinkAddress>();
1100            var queryResultFiller = new ListFiller<TLinkAddress, TLinkAddress>(queryResult,
                ↪ links.Constants.Continue);
1101            links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
1102            foreach (var link in queryResult)
1103            {
1104                links.Delete(link);
1105            }
1106        }
1107
1108        // TODO: Move to Platform.Data
1109        /// <summary>
```

```
1110            /// <para>
1111            /// Determines whether are values reset.
1112            /// </para>
1113            /// <para></para>
1114            /// </summary>
1115            /// <typeparam name="TLinkAddress">
1116            /// <para>The link.</para>
1117            /// <para></para>
1118            /// </typeparam>
1119            /// <param name="links">
1120            /// <para>The links.</para>
1121            /// <para></para>
1122            /// </param>
1123            /// <param name="linkIndex">
1124            /// <para>The link index.</para>
1125            /// <para></para>
1126            /// </param>
1127            /// <returns>
1128            /// <para>The bool</para>
1129            /// <para></para>
1130            /// </returns>
1131            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1132            public static bool AreValuesReset<TLinkAddress>(this ILinks<TLinkAddress> links,
                ↪ TLinkAddress linkIndex)
1133            {
1134                var nullConstant = links.Constants.Null;
1135                var equalityComparer = EqualityComparer<TLinkAddress>.Default;
1136                var link = links.GetLink(linkIndex);
1137                for (int i = 1; i < link.Count; i++)
1138                {
1139                    if (!equalityComparer.Equals(link[i], nullConstant))
1140                    {
1141                        return false;
1142                    }
1143                }
1144                return true;
1145            }
1146
1147            public static void ResetValues<TLinkAddress>(this ILinks<TLinkAddress> links,
                ↪ TLinkAddress linkIndex)  => links.ResetValues(linkIndex, null);
1148
1149            // TODO: Create a universal version of this method in Platform.Data (with using of for
                ↪ loop)
1150            /// <summary>
1151            /// <para>
1152            /// Resets the values using the specified links.
1153            /// </para>
1154            /// <para></para>
1155            /// </summary>
1156            /// <typeparam name="TLinkAddress">
1157            /// <para>The link.</para>
1158            /// <para></para>
1159            /// </typeparam>
1160            /// <param name="links">
1161            /// <para>The links.</para>
1162            /// <para></para>
1163            /// </param>
1164            /// <param name="linkIndex">
1165            /// <para>The link index.</para>
1166            /// <para></para>
1167            /// </param>
1168            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1169            public static TLinkAddress ResetValues<TLinkAddress>(this ILinks<TLinkAddress> links,
                ↪ TLinkAddress linkIndex, WriteHandler<TLinkAddress>? handler)
1170            {
1171                var nullConstant = links.Constants.Null;
1172                var updateRequest = new Link<TLinkAddress>(linkIndex, nullConstant, nullConstant);
1173                return links.Update(updateRequest, handler);
1174            }
1175
1176            public static void EnforceResetValues<TLinkAddress>(this ILinks<TLinkAddress> links,
                ↪ TLinkAddress linkIndex)  => links.EnforceResetValues(linkIndex, null);
1177
1178
1179            // TODO: Create a universal version of this method in Platform.Data (with using of for
                ↪ loop)
1180            /// <summary>
1181            /// <para>
```

```
1182            /// Enforces the reset values using the specified links.
1183            /// </para>
1184            /// <para></para>
1185            /// </summary>
1186            /// <typeparam name="TLinkAddress">
1187            /// <para>The link.</para>
1188            /// <para></para>
1189            /// </typeparam>
1190            /// <param name="links">
1191            /// <para>The links.</para>
1192            /// <para></para>
1193            /// </param>
1194            /// <param name="linkIndex">
1195            /// <para>The link index.</para>
1196            /// <para></para>
1197            /// </param>
1198            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1199            public static TLinkAddress EnforceResetValues<TLinkAddress>(this ILinks<TLinkAddress>
                ↪ links, TLinkAddress linkIndex, WriteHandler<TLinkAddress>? handler)
1200            {
1201                if (!links.AreValuesReset(linkIndex))
1202                {
1203                    return links.ResetValues(linkIndex, handler);
1204                }
1205                return links.Constants.Continue;
1206            }
1207
1208            public static void MergeUsages<TLinkAddress>(this ILinks<TLinkAddress> links,
                ↪ TLinkAddress oldLinkIndex, TLinkAddress newLinkIndex)  =>
                ↪ links.MergeUsages(oldLinkIndex, newLinkIndex, null);
1209
1210            /// <summary>
1211            /// Merging two usages graphs, all children of old link moved to be children of new link
                ↪ or deleted.
1212            /// </summary>
1213            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1214            public static TLinkAddress MergeUsages<TLinkAddress>(this ILinks<TLinkAddress> links,
                ↪ TLinkAddress oldLinkIndex, TLinkAddress newLinkIndex, WriteHandler<TLinkAddress>?
                ↪ handler)
1215            {
1216                var equalityComparer = EqualityComparer<TLinkAddress>.Default;
1217                if (equalityComparer.Equals(oldLinkIndex, newLinkIndex))
1218                {
1219                    return newLinkIndex;
1220                }
1221                var constants = links.Constants;
1222                var usagesAsSource = links.All(new Link<TLinkAddress>(constants.Any, oldLinkIndex,
                    ↪ constants.Any));
1223                WriteHandlerState<TLinkAddress> handlerState = new(constants.Continue,
                    ↪ constants.Break, handler);
1224                for (var i = 0; i < usagesAsSource.Count; i++)
1225                {
1226                    var usageAsSource = usagesAsSource[i];
1227                    if (equalityComparer.Equals(links.GetIndex(usageAsSource), oldLinkIndex))
1228                    {
1229                        continue;
1230                    }
1231                    var restriction = new LinkAddress<TLinkAddress>(links.GetIndex(usageAsSource));
1232                    var substitution = new Link<TLinkAddress>(newLinkIndex,
                        ↪ links.GetTarget(usageAsSource));
1233                    handlerState.Apply(links.Update(restriction, substitution,
                        ↪ handlerState.Handler));
1234                }
1235                var usagesAsTarget = links.All(new Link<TLinkAddress>(constants.Any, constants.Any,
                    ↪ oldLinkIndex));
1236                for (var i = 0; i < usagesAsTarget.Count; i++)
1237                {
1238                    var usageAsTarget = usagesAsTarget[i];
1239                    if (equalityComparer.Equals(links.GetIndex(usageAsTarget), oldLinkIndex))
1240                    {
1241                        continue;
1242                    }
1243                    var restriction = links.GetLink(links.GetIndex(usageAsTarget));
1244                    var substitution = new Link<TLinkAddress>(links.GetTarget(usageAsTarget),
                        ↪ newLinkIndex);
1245                    handlerState.Apply(links.Update(restriction, substitution,
                        ↪ handlerState.Handler));
1246                }
```

```
1247            return handlerState.Result;
1248        }
1249
1250        public static TLinkAddress MergeAndDelete<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪ TLinkAddress oldLinkIndex, TLinkAddress newLinkIndex)
1251        {
1252            var equalityComparer = EqualityComparer<TLinkAddress>.Default;
1253            if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
1254            {
1255                links.MergeUsages(oldLinkIndex, newLinkIndex);
1256                links.Delete(oldLinkIndex);
1257            }
1258            return newLinkIndex;
1259        }
1260
1261        /// <summary>
1262        /// Replace one link with another (replaced link is deleted, children are updated or
            ↪ deleted).
1263        /// </summary>
1264        [MethodImpl(MethodImplOptions.AggressiveInlining)]
1265        public static TLinkAddress MergeAndDelete<TLinkAddress>(this ILinks<TLinkAddress> links,
            ↪ TLinkAddress oldLinkIndex, TLinkAddress newLinkIndex, WriteHandler<TLinkAddress>?
            ↪ handler)
1266        {
1267            var equalityComparer = EqualityComparer<TLinkAddress>.Default;
1268            var constants = links.Constants;
1269            WriteHandlerState<TLinkAddress> handlerState = new(constants.Continue,
                ↪ constants.Break, handler);
1270            if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
1271            {
1272                handlerState.Apply(links.MergeUsages(oldLinkIndex, newLinkIndex,
                    ↪ handlerState.Handler));
1273                handlerState.Apply(links.Delete(oldLinkIndex, handlerState.Handler));
1274            }
1275            return handlerState.Result;
1276        }
1277
1278        /// <summary>
1279        /// <para>
1280        /// Decorates the with automatic uniqueness and usages resolution using the specified
            ↪ links.
1281        /// </para>
1282        /// <para></para>
1283        /// </summary>
1284        /// <typeparam name="TLinkAddress">
1285        /// <para>The link.</para>
1286        /// <para></para>
1287        /// </typeparam>
1288        /// <param name="links">
1289        /// <para>The links.</para>
1290        /// <para></para>
1291        /// </param>
1292        /// <returns>
1293        /// <para>The links.</para>
1294        /// <para></para>
1295        /// </returns>
1296        [MethodImpl(MethodImplOptions.AggressiveInlining)]
1297        public static ILinks<TLinkAddress>
            ↪ DecorateWithAutomaticUniquenessAndUsagesResolution<TLinkAddress>(this
            ↪ ILinks<TLinkAddress> links)
1298        {
1299            links = new LinksCascadeUsagesResolver<TLinkAddress>(links);
1300            links = new NonNullContentsLinkDeletionResolver<TLinkAddress>(links);
1301            links = new LinksCascadeUniquenessAndUsagesResolver<TLinkAddress>(links);
1302            return links;
1303        }
1304
1305        /// <summary>
1306        /// <para>
1307        /// Formats the links.
1308        /// </para>
1309        /// <para></para>
1310        /// </summary>
1311        /// <typeparam name="TLinkAddress">
1312        /// <para>The link.</para>
1313        /// <para></para>
1314        /// </typeparam>
1315        /// <param name="links">
```

```
1316          /// <para>The links.</para>
1317          /// <para></para>
1318          /// </param>
1319          /// <param name="link">
1320          /// <para>The link.</para>
1321          /// <para></para>
1322          /// </param>
1323          /// <returns>
1324          /// <para>The string</para>
1325          /// <para></para>
1326          /// </returns>
1327          [MethodImpl(MethodImplOptions.AggressiveInlining)]
1328          public static string Format<TLinkAddress>(this ILinks<TLinkAddress> links,
              ↪ IList<TLinkAddress>? link)
1329          {
1330              var constants = links.Constants;
1331              return $"({links.GetIndex(link)}: {links.GetSource(link)} {links.GetTarget(link)})";
1332          }
1333
1334          /// <summary>
1335          /// <para>
1336          /// Formats the links.
1337          /// </para>
1338          /// <para></para>
1339          /// </summary>
1340          /// <typeparam name="TLinkAddress">
1341          /// <para>The link.</para>
1342          /// <para></para>
1343          /// </typeparam>
1344          /// <param name="links">
1345          /// <para>The links.</para>
1346          /// <para></para>
1347          /// </param>
1348          /// <param name="link">
1349          /// <para>The link.</para>
1350          /// <para></para>
1351          /// </param>
1352          /// <returns>
1353          /// <para>The string</para>
1354          /// <para></para>
1355          /// </returns>
1356          [MethodImpl(MethodImplOptions.AggressiveInlining)]
1357          public static string Format<TLinkAddress>(this ILinks<TLinkAddress> links, TLinkAddress
              ↪ link)  => links.Format(links.GetLink(link));
1358      }
1359  }
```

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

```
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Data.Doublets
4  {
5      /// <summary>
6      /// <para>
7      /// Defines the synchronized links.
8      /// </para>
9      /// <para></para>
10     /// </summary>
11     /// <seealso cref="ISynchronizedLinks{TLinkAddress, ILinks{TLinkAddress},
       ↪ LinksConstants{TLinkAddress}}"/>
12     /// <seealso cref="ILinks{TLinkAddress}"/>
13     public interface ISynchronizedLinks<TLinkAddress> : ISynchronizedLinks<TLinkAddress,
       ↪ ILinks<TLinkAddress>, LinksConstants<TLinkAddress>>, ILinks<TLinkAddress>
14     {
15     }
16 }
```

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

```
1  using Platform.Collections.Lists;
2  using Platform.Exceptions;
3  using Platform.Ranges;
4  using Platform.Singletons;
5  using System;
6  using System.Collections;
7  using System.Collections.Generic;
8  using System.Runtime.CompilerServices;
9
10 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
```

```
11
12  namespace Platform.Data.Doublets
13  {
14      /// <summary>
15      /// Структура описывающая уникальную связь.
16      /// </summary>
17      public struct Link<TLinkAddress> : IEquatable<Link<TLinkAddress>>,
        ↪  IReadOnlyList<TLinkAddress>, IList<TLinkAddress>
18      {
19          /// <summary>
20          /// <para>
21          /// The link.
22          /// </para>
23          /// <para></para>
24          /// </summary>
25          public static readonly Link<TLinkAddress> Null = new Link<TLinkAddress>();
26          private static readonly LinksConstants<TLinkAddress> _constants =
            ↪  Default<LinksConstants<TLinkAddress>>.Instance;
27          private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
            ↪  EqualityComparer<TLinkAddress>.Default;
28          private const int Length = 3;
29
30          /// <summary>
31          /// <para>
32          /// The index.
33          /// </para>
34          /// <para></para>
35          /// </summary>
36          public readonly TLinkAddress Index;
37          /// <summary>
38          /// <para>
39          /// The source.
40          /// </para>
41          /// <para></para>
42          /// </summary>
43          public readonly TLinkAddress Source;
44          /// <summary>
45          /// <para>
46          /// The target.
47          /// </para>
48          /// <para></para>
49          /// </summary>
50          public readonly TLinkAddress Target;
51
52          /// <summary>
53          /// <para>
54          /// Initializes a new <see cref="Link"/> instance.
55          /// </para>
56          /// <para></para>
57          /// </summary>
58          /// <param name="values">
59          /// <para>A values.</para>
60          /// <para></para>
61          /// </param>
62          [MethodImpl(MethodImplOptions.AggressiveInlining)]
63          public Link(params TLinkAddress[] values) => SetValues(values, out Index, out Source,
            ↪  out Target);
64
65          /// <summary>
66          /// <para>
67          /// Initializes a new <see cref="Link"/> instance.
68          /// </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>
```

```
        /// <param name="other">
        /// <para>A other.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="NotSupportedException">
        /// <para></para>
        /// <para></para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Link(object other)
        {
            if (other is Link<TLinkAddress> otherLink)
            {
                SetValues(ref otherLink, out Index, out Source, out Target);
            }
            else if(other is IList<TLinkAddress> otherList)
            {
                SetValues(otherList, out Index, out Source, out Target);
            }
            else
            {
                throw new NotSupportedException();
            }
        }

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

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Link"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="index">
        /// <para>A index.</para>
        /// <para></para>
        /// </param>
        /// <param name="source">
        /// <para>A source.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>A target.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Link(TLinkAddress index, TLinkAddress source, TLinkAddress target)
        {
            Index = index;
            Source = source;
            Target = target;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void SetValues(ref Link<TLinkAddress> other, out TLinkAddress index, out
        ↪  TLinkAddress source, out TLinkAddress target)
        {
            index = other.Index;
            source = other.Source;
            target = other.Target;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void SetValues(IList<TLinkAddress>? values, out TLinkAddress index, out
        ↪  TLinkAddress source, out TLinkAddress target)
        {
            if (values == null)
            {
```

```
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.
```

```
        /// </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(Link<TLinkAddress> other) => _equalityComparer.Equals(Index,
        ↪  other.Index)
                                                && _equalityComparer.Equals(Source, other.Source)
                                                && _equalityComparer.Equals(Target, other.Target);

        /// <summary>
        /// <para>
        /// Returns the string using the specified index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="index">
        /// <para>The index.</para>
        /// <para></para>
        /// </param>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string ToString(TLinkAddress index, TLinkAddress source, TLinkAddress
        ↪  target) => $"({index}: {source}->{target})";

        /// <summary>
        /// <para>
        /// Returns the string using the specified 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 string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string ToString(TLinkAddress source, TLinkAddress target) =>
        ↪  $"({source}->{target})";

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

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

        /// <summary>
        /// <para>
        /// Returns the string.
        /// </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
369             }
370             [MethodImpl(MethodImplOptions.AggressiveInlining)]
371             set => throw new NotSupportedException();
372         }
373
374         /// <summary>
375         /// <para>
376         /// Gets the enumerator.
377         /// </para>
378         /// <para></para>
379         /// </summary>
380         /// <returns>
381         /// <para>The enumerator</para>
382         /// <para></para>
383         /// </returns>
384         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
385         IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
386 
387         /// <summary>
388         /// <para>
389         /// Gets the enumerator.
390         /// </para>
391         /// <para></para>
392         /// </summary>
393         /// <returns>
394         /// <para>An enumerator of t link</para>
395         /// <para></para>
396         /// </returns>
397         [MethodImpl(MethodImplOptions.AggressiveInlining)]
398         public IEnumerator<TLinkAddress> GetEnumerator()
399         {
400             yield return Index;
401             yield return Source;
402             yield return Target;
403         }
404 
405         /// <summary>
406         /// <para>
407         /// Adds the item.
408         /// </para>
409         /// <para></para>
410         /// </summary>
411         /// <param name="item">
412         /// <para>The item.</para>
413         /// <para></para>
414         /// </param>
415         [MethodImpl(MethodImplOptions.AggressiveInlining)]
416         public void Add(TLinkAddress item) => throw new NotSupportedException();
417 
418         /// <summary>
419         /// <para>
420         /// Clears this instance.
421         /// </para>
422         /// <para></para>
423         /// </summary>
424         [MethodImpl(MethodImplOptions.AggressiveInlining)]
425         public void Clear() => throw new NotSupportedException();
426 
427         /// <summary>
428         /// <para>
429         /// Determines whether this instance contains.
430         /// </para>
431         /// <para></para>
432         /// </summary>
433         /// <param name="item">
434         /// <para>The item.</para>
435         /// <para></para>
436         /// </param>
437         /// <returns>
438         /// <para>The bool</para>
439         /// <para></para>
440         /// </returns>
441         [MethodImpl(MethodImplOptions.AggressiveInlining)]
442         public bool Contains(TLinkAddress item) => IndexOf(item) >= 0;
443 
444         /// <summary>
445         /// <para>
446         /// Copies the to using the specified array.
447         /// </para>
448         /// <para></para>
449         /// </summary>
450         /// <param name="array">
451         /// <para>The array.</para>
452         /// <para></para>
453         /// </param>
454         /// <param name="arrayIndex">
455         /// <para>The array index.</para>
456         /// <para></para>
457         /// </param>
458         /// <exception cref="InvalidOperationException">
459         /// <para></para>
460         /// <para></para>
461         /// </exception>
462         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
463         public void CopyTo(TLinkAddress[] array, int arrayIndex)
464         {
465             Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
466             Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
                ↪  nameof(arrayIndex));
467             if (arrayIndex + Length > array.Length)
468             {
469                 throw new InvalidOperationException();
470             }
471             array[arrayIndex++] = Index;
472             array[arrayIndex++] = Source;
473             array[arrayIndex] = Target;
474         }
475 
476         /// <summary>
477         /// <para>
478         /// Determines whether this instance remove.
479         /// </para>
480         /// <para></para>
481         /// </summary>
482         /// <param name="item">
483         /// <para>The item.</para>
484         /// <para></para>
485         /// </param>
486         /// <returns>
487         /// <para>The bool</para>
488         /// <para></para>
489         /// </returns>
490         [MethodImpl(MethodImplOptions.AggressiveInlining)]
491         public bool Remove(TLinkAddress item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
492 
493         /// <summary>
494         /// <para>
495         /// Indexes the of using the specified item.
496         /// </para>
497         /// <para></para>
498         /// </summary>
499         /// <param name="item">
500         /// <para>The item.</para>
501         /// <para></para>
502         /// </param>
503         /// <returns>
504         /// <para>The int</para>
505         /// <para></para>
506         /// </returns>
507         [MethodImpl(MethodImplOptions.AggressiveInlining)]
508         public int IndexOf(TLinkAddress item)
509         {
510             if (_equalityComparer.Equals(Index, item))
511             {
512                 return _constants.IndexPart;
513             }
514             if (_equalityComparer.Equals(Source, item))
515             {
516                 return _constants.SourcePart;
517             }
518             if (_equalityComparer.Equals(Target, item))
519             {
520                 return _constants.TargetPart;
521             }
522             return -1;
523         }
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)]
```

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

        /// <summary>
        /// <para>
        /// Removes the at using the specified index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="index">
        /// <para>The index.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void RemoveAt(int index) => throw new NotSupportedException();

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

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

        #endregion
    }
}
```

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

```
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets
{
    /// <summary>
    /// <para>
    /// Represents the link extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class LinkExtensions
    {
        /// <summary>
        /// <para>
        /// Determines whether is full 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 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>
```

```
50          /// <returns>
51          /// <para>The bool</para>
52          /// <para></para>
53          /// </returns>
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          public static bool IsPartialPoint<TLinkAddress>(this Link<TLinkAddress> link) =>
            ↪  Point<TLinkAddress>.IsPartialPoint(link);
56      }
57  }
```

1.28 ./csharp/Platform.Data.Doublets/LinksOperatorBase.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 links operator base.
10      /// </para>
11      /// <para></para>
12      /// </summary>
13      public abstract class LinksOperatorBase<TLinkAddress>
14      {
15          /// <summary>
16          /// <para>
17          /// The links.
18          /// </para>
19          /// <para></para>
20          /// </summary>
21          protected readonly ILinks<TLinkAddress> _links;
22
23          /// <summary>
24          /// <para>
25          /// Gets the links value.
26          /// </para>
27          /// <para></para>
28          /// </summary>
29          public ILinks<TLinkAddress> Links
30          {
31              [MethodImpl(MethodImplOptions.AggressiveInlining)]
32              get => _links;
33          }
34
35          /// <summary>
36          /// <para>
37          /// Initializes a new <see cref="LinksOperatorBase"/> instance.
38          /// </para>
39          /// <para></para>
40          /// </summary>
41          /// <param name="links">
42          /// <para>A links.</para>
43          /// <para></para>
44          /// </param>
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected LinksOperatorBase(ILinks<TLinkAddress> links) => _links = links;
47      }
48  }
```

1.29 ./csharp/Platform.Data.Doublets/Memory/ILinksListMethods.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
6   {
7       /// <summary>
8       /// <para>
9       /// Defines the links list methods.
10      /// </para>
11      /// <para></para>
12      /// </summary>
13      public interface ILinksListMethods<TLinkAddress>
14      {
15          /// <summary>
16          /// <para>
17          /// 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);
    }
}
```

1.30 ./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);
    }
}
```

1.31 ./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
    }
}
```

1.32 ./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);
    }
}
```

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

```
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>
```

```
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>
200         /// </summary>
201         /// <param name="link">
202         /// <para>The link.</para>
203         /// <para></para>
204         /// </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)
        {
            ref var firstLink = ref GetLinkDataPartReference(first);
            ref var secondLink = ref GetLinkDataPartReference(second);
            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();
```

```
            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>
```

```
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/ExternalLinksSizeBalancedTreeMethodsBase.cs

```
1   using System;
2   using System.Text;
3   using System.Collections.Generic;
4   using System.Runtime.CompilerServices;
5   using Platform.Collections.Methods.Trees;
6   using Platform.Converters;
7   using 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.Split.Generic
13  {
14      /// <summary>
15      /// <para>
16      /// Represents the external links size balanced tree methods base.
17      /// </para>
18      /// <para></para>
19      /// </summary>
20      /// <seealso cref="SizeBalancedTreeMethods{TLinkAddress}"/>
21      /// <seealso cref="ILinksTreeMethods{TLinkAddress}"/>
22      public unsafe abstract class ExternalLinksSizeBalancedTreeMethodsBase<TLinkAddress> :
        ↪  SizeBalancedTreeMethods<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 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>
```

```
        /// 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>
        /// <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>
        /// <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 GetLinkDataPartReference(first);
            ref var secondLink = ref GetLinkDataPartReference(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>
```

```
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                     }
```

```
                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);
        }
```

```
    }
}
```

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

```
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Memory.Split.Generic
{
    /// <summary>
    /// <para>
    /// Represents the external links sources recursionless size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ExternalLinksRecursionlessSizeBalancedTreeMethodsBase{TLinkAddress}"/>
    public unsafe class ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods<TLinkAddress> :
    ↪ ExternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see
        ↪ cref="ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods"/> 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 ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(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>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <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>
```

```
        /// <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 GetLinkIndexPartReference(node);
            link.LeftAsSource = Zero;
            link.RightAsSource = Zero;
            link.SizeAsSource = Zero;
        }
    }
}
```

1.36 ./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>
        /// </param>
        /// <param name="right">
        /// <para>The right.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
        ↪ GetLinkIndexPartReference(node).RightAsSource = right;

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

```
        /// <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 GetLinkIndexPartReference(node);
            link.LeftAsSource = Zero;
            link.RightAsSource = Zero;
            link.SizeAsSource = Zero;
        }
    }
}
```

1.37 ./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>
        /// </param>
        /// <returns>
        /// <para>The ref link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
        ↪  GetLinkIndexPartReference(node).RightAsTarget;

        /// <summary>
        /// <para>
        /// Gets the left using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <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 }
```

1.38 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksTargetsSizeBalancedTreeMethods.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 external links targets size balanced tree methods.
10     /// </para>
11     /// <para></para>
12     /// </summary>
13     /// <seealso cref="ExternalLinksSizeBalancedTreeMethodsBase{TLinkAddress}"/>
14     public unsafe class ExternalLinksTargetsSizeBalancedTreeMethods<TLinkAddress> :
       ↪ ExternalLinksSizeBalancedTreeMethodsBase<TLinkAddress>
15     {
```

```
        /// <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)]
        protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
        ↪  GetLinkIndexPartReference(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>
```

```
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>
```

```
        /// <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>
        /// </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)]
```

```
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 }
```

1.39 ./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;
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.Split.Generic
13 {
14     /// <summary>
15     /// <para>
16     /// Represents the internal links recursionless size balanced tree methods base.
17     /// </para>
18     /// <para></para>
19     /// </summary>
20     /// <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">
        /// <para>A header.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected
        ↪  InternalLinksRecursionlessSizeBalancedTreeMethodsBase(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 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 GetTreeRoot(TLinkAddress link);

        /// <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>
```

```
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>
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 list of t link</para>
233         /// <para></para>
234         /// </returns>
235         [MethodImpl(MethodImplOptions.AggressiveInlining)]
236         protected virtual IList<TLinkAddress>? GetLinkValues(TLinkAddress linkIndex)
237         {
238             ref var link = ref GetLinkDataPartReference(linkIndex);
239             return new Link<TLinkAddress>(linkIndex, link.Source, link.Target);
```

```
240         }
241
242         /// <summary>
243         /// <para>
244         /// The zero.
245         /// </para>
246         /// <para></para>
247         /// </summary>
248         public TLinkAddress this[TLinkAddress link, TLinkAddress index]
249         {
250             [MethodImpl(MethodImplOptions.AggressiveInlining)]
251             get
252             {
253                 var root = GetTreeRoot(link);
254                 if (GreaterOrEqualThan(index, GetSize(root)))
255                 {
256                     return Zero;
257                 }
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 
```

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

1.40 ./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>
        /// 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="InternalLinksSizeBalancedTreeMethodsBase"/> 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 InternalLinksSizeBalancedTreeMethodsBase(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 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 GetTreeRoot(TLinkAddress link);

        /// <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);
```

```
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>
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 list of t link</para>
233             /// <para></para>
234             /// </returns>
235             [MethodImpl(MethodImplOptions.AggressiveInlining)]
236             protected virtual IList<TLinkAddress>? GetLinkValues(TLinkAddress linkIndex)
237             {
238                 ref var link = ref GetLinkDataPartReference(linkIndex);
239                 return new Link<TLinkAddress>(linkIndex, link.Source, link.Target);
240             }
241
242             /// <summary>
243             /// <para>
244             /// The zero.
245             /// </para>
246             /// <para></para>
247             /// </summary>
248             public TLinkAddress this[TLinkAddress link, TLinkAddress index]
249             {
250                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
251                 get
252                 {
253                     var root = GetTreeRoot(link);
254                     if (GreaterOrEqualThan(index, GetSize(root)))
255                     {
256                         return Zero;
257                     }
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));
```

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

1.41 ./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>
```

```
151             /// </summary>
152             /// <param name="element">
153             /// <para>The element.</para>
154             /// <para></para>
155             /// </param>
156             /// <returns>
157             /// <para>The link</para>
158             /// <para></para>
159             /// </returns>
160             [MethodImpl(MethodImplOptions.AggressiveInlining)]
161             protected override TLinkAddress GetPrevious(TLinkAddress element) =>
                ↪ GetLinkIndexPartReference(element).LeftAsSource;
162
163             /// <summary>
164             /// <para>
165             /// Gets the next using the specified element.
166             /// </para>
167             /// <para></para>
168             /// </summary>
169             /// <param name="element">
170             /// <para>The element.</para>
171             /// <para></para>
172             /// </param>
173             /// <returns>
174             /// <para>The link</para>
175             /// <para></para>
176             /// </returns>
177             [MethodImpl(MethodImplOptions.AggressiveInlining)]
178             protected override TLinkAddress GetNext(TLinkAddress element) =>
                ↪ GetLinkIndexPartReference(element).RightAsSource;
179
180             /// <summary>
181             /// <para>
182             /// Gets the size using the specified head.
183             /// </para>
184             /// <para></para>
185             /// </summary>
186             /// <param name="head">
187             /// <para>The head.</para>
188             /// <para></para>
189             /// </param>
190             /// <returns>
191             /// <para>The link</para>
192             /// <para></para>
193             /// </returns>
194             [MethodImpl(MethodImplOptions.AggressiveInlining)]
195             protected override TLinkAddress GetSize(TLinkAddress head) =>
                ↪ GetLinkIndexPartReference(head).SizeAsSource;
196
197             /// <summary>
198             /// <para>
199             /// Sets the first using the specified head.
200             /// </para>
201             /// <para></para>
202             /// </summary>
203             /// <param name="head">
204             /// <para>The head.</para>
205             /// <para></para>
206             /// </param>
207             /// <param name="element">
208             /// <para>The element.</para>
209             /// <para></para>
210             /// </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>
286             /// <param name="size">
287             /// <para>The size.</para>
288             /// <para></para>
289             /// </param>
290             [MethodImpl(MethodImplOptions.AggressiveInlining)]
291             protected override void SetSize(TLinkAddress head, TLinkAddress size) =>
                 ↪ GetLinkIndexPartReference(head).SizeAsSource = size;
292
293             /// <summary>
294             /// <para>
295             /// Counts the usages using the specified head.
296             /// </para>
297             /// <para></para>
298             /// </summary>
299             /// <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;
        }
    }
}
```

1.42 ./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>
        /// </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>
        /// </param>
        /// <param name="right">
        /// <para>The right.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(TLinkAddress node, TLinkAddress right) =>
        ↪  GetLinkIndexPartReference(node).RightAsSource = right;

        /// <summary>
        /// <para>
        /// Gets the size using the specified node.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <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;
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).Target;
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.LeftAsSource = Zero;
243             link.RightAsSource = Zero;
244             link.SizeAsSource = 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(source), target);
266     }
267 }
```

1.43 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSourcesSizeBalancedTreeMethods.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 sources size balanced tree methods.
10     /// </para>
11     /// <para></para>
12     /// </summary>
13     /// <seealso cref="InternalLinksSizeBalancedTreeMethodsBase{TLinkAddress}"/>
14     public unsafe class InternalLinksSourcesSizeBalancedTreeMethods<TLinkAddress> :
       ↪ InternalLinksSizeBalancedTreeMethodsBase<TLinkAddress>
15     {
16         /// <summary>
17         /// <para>
18         /// Initializes a new <see cref="InternalLinksSourcesSizeBalancedTreeMethods"/> 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">
```

```
        /// <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>
        [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>
```

```
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;
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).Target;
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.LeftAsSource = Zero;
243             link.RightAsSource = Zero;
244             link.SizeAsSource = Zero;
245         }
246
247         /// <summary>
248         /// <para>
249         /// Searches the source.
250         /// </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);
    }
}
```

1.44 ./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)]
        public InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(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>
```

```
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>
```

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

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

```
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Memory.Split.Generic
{
```

```
    /// <summary>
    /// <para>
    /// Represents the internal links targets size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="InternalLinksSizeBalancedTreeMethodsBase{TLinkAddress}"/>
    public unsafe class InternalLinksTargetsSizeBalancedTreeMethods<TLinkAddress> :
    ↪  InternalLinksSizeBalancedTreeMethodsBase<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="InternalLinksTargetsSizeBalancedTreeMethods"/> 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 InternalLinksTargetsSizeBalancedTreeMethods(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)]
        protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
        ↪  GetLinkIndexPartReference(node).RightAsTarget;

        /// <summary>
        /// <para>
        /// Gets the left using the specified node.
        /// </para>
        /// <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;
```

```

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

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

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

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

namespace Platform.Data.Doublets.Memory.Split.Generic
{
    /// <summary>
    /// <para>
    /// Represents the split memory links.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="SplitMemoryLinksBase{TLinkAddress}"/>
    public unsafe class SplitMemoryLinks<TLinkAddress> : SplitMemoryLinksBase<TLinkAddress>
    {
        private readonly Func<ILinksTreeMethods<TLinkAddress>> _createInternalSourceTreeMethods;
        private readonly Func<ILinksTreeMethods<TLinkAddress>> _createExternalSourceTreeMethods;
        private readonly Func<ILinksTreeMethods<TLinkAddress>> _createInternalTargetTreeMethods;
        private readonly Func<ILinksTreeMethods<TLinkAddress>> _createExternalTargetTreeMethods;
        private byte* _header;
        private byte* _linksDataParts;
        private byte* _linksIndexParts;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="SplitMemoryLinks"/> 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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SplitMemoryLinks(string dataMemory, string indexMemory) : this(new
        ↪  FileMappedResizableDirectMemory(dataMemory), new
        ↪  FileMappedResizableDirectMemory(indexMemory)) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="SplitMemoryLinks"/> 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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
        ↪  indexMemory) : this(dataMemory, indexMemory, DefaultLinksSizeStep) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="SplitMemoryLinks"/> 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)]
        public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
        ↪  indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
        ↪  memoryReservationStep, Default<LinksConstants<TLinkAddress>>.Instance,
        ↪  IndexTreeType.Default, useLinkedList: true) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="SplitMemoryLinks"/> 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>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        [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.
```

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

        /// <summary>
        /// <para>
        /// Resets the pointers.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ResetPointers()
        {
            base.ResetPointers();
            _linksDataParts = null;
            _linksIndexParts = 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 data part 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 data part of t link</para>
        /// <para></para>
        /// </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 }
```

1.47 ./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>
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>
181             /// <para></para>
182             /// </param>
183             /// <param name="memoryReservationStep">
184             /// <para>A memory reservation step.</para>
185             /// <para></para>
186             /// </param>
187             /// <param name="constants">
188             /// <para>A constants.</para>
189             /// <para></para>
190             /// </param>
191             /// <param name="useLinkedList">
192             /// <para>A use linked list.</para>
193             /// <para></para>
194             /// </param>
195             [MethodImpl(MethodImplOptions.AggressiveInlining)]
196             protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
                ↪  indexMemory, long memoryReservationStep, LinksConstants<TLinkAddress> constants,
                ↪  bool useLinkedList)
```

```
197         {
198             _dataMemory = dataMemory;
199             _indexMemory = indexMemory;
200             _dataMemoryReservationStepInBytes = memoryReservationStep * LinkDataPartSizeInBytes;
201             _indexMemoryReservationStepInBytes = memoryReservationStep *
                ↪  LinkIndexPartSizeInBytes;
202             _useLinkedList = useLinkedList;
203             Constants = constants;
204         }
205 
206         /// <summary>
207         /// <para>
208         /// Initializes a new <see cref="SplitMemoryLinksBase"/> instance.
209         /// </para>
210         /// <para></para>
211         /// </summary>
212         /// <param name="dataMemory">
213         /// <para>A data memory.</para>
214         /// <para></para>
215         /// </param>
216         /// <param name="indexMemory">
217         /// <para>A index memory.</para>
218         /// <para></para>
219         /// </param>
220         /// <param name="memoryReservationStep">
221         /// <para>A memory reservation step.</para>
222         /// <para></para>
223         /// </param>
224         [MethodImpl(MethodImplOptions.AggressiveInlining)]
225         protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪  indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
            ↪  memoryReservationStep, Default<LinksConstants<TLinkAddress>>.Instance,
            ↪  useLinkedList: true) { }
226 
227         /// <summary>
228         /// <para>
229         /// Inits the data memory.
230         /// </para>
231         /// <para></para>
232         /// </summary>
233         /// <param name="dataMemory">
234         /// <para>The data memory.</para>
235         /// <para></para>
236         /// </param>
237         /// <param name="indexMemory">
238         /// <para>The index memory.</para>
239         /// <para></para>
240         /// </param>
241         [MethodImpl(MethodImplOptions.AggressiveInlining)]
242         protected virtual void Init(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪  indexMemory)
243         {
244             // Read allocated links from header
245             if (indexMemory.ReservedCapacity < LinkHeaderSizeInBytes)
246             {
247                 indexMemory.ReservedCapacity = LinkHeaderSizeInBytes;
248             }
249             SetPointers(dataMemory, indexMemory);
250             ref var header = ref GetHeaderReference();
251             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             }
```

```
            // Check for alignment
            if (minimumDataReservedCapacity % _dataMemoryReservationStepInBytes > 0)
            {
                minimumDataReservedCapacity = ((minimumDataReservedCapacity /
                ↪  _dataMemoryReservationStepInBytes) * _dataMemoryReservationStepInBytes) +
                ↪  _dataMemoryReservationStepInBytes;
            }
            if (minimumIndexReservedCapacity % _indexMemoryReservationStepInBytes > 0)
            {
                minimumIndexReservedCapacity = ((minimumIndexReservedCapacity /
                ↪  _indexMemoryReservationStepInBytes) * _indexMemoryReservationStepInBytes) +
                ↪  _indexMemoryReservationStepInBytes;
            }
            if (dataMemory.ReservedCapacity != minimumDataReservedCapacity)
            {
                dataMemory.ReservedCapacity = minimumDataReservedCapacity;
            }
            if (indexMemory.ReservedCapacity != minimumIndexReservedCapacity)
            {
                indexMemory.ReservedCapacity = minimumIndexReservedCapacity;
            }
            SetPointers(dataMemory, indexMemory);
            header = ref GetHeaderReference();
            // Ensure correctness _memory.UsedCapacity over _header->AllocatedLinks
            // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
            dataMemory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) *
            ↪  LinkDataPartSizeInBytes) + LinkDataPartSizeInBytes; // First link is read only
            ↪  zero link.
            indexMemory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) *
            ↪  LinkIndexPartSizeInBytes) + LinkHeaderSizeInBytes;
            // Ensure correctness _memory.ReservedLinks over _header->ReservedCapacity
            // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
            header.ReservedLinks = ConvertToAddress((dataMemory.ReservedCapacity -
            ↪  LinkDataPartSizeInBytes) / LinkDataPartSizeInBytes);
        }

        /// <summary>
        /// <para>
        /// Counts the substitution.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <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 Count(IList<TLinkAddress>? restriction)
        {
            // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
            if (restriction.Count == 0)
            {
                return Total;
            }
            var constants = Constants;
            var any = constants.Any;
            var index = this.GetIndex(restriction);
            if (restriction.Count == 1)
            {
                if (AreEqual(index, any))
                {
                    return Total;
                }
                return Exists(index) ? GetOne() : GetZero();
            }
            if (restriction.Count == 2)
            {
                var value = restriction[1];
                if (AreEqual(index, any))
                {
```

```
                    if (AreEqual(value, any))
                    {
                        return Total; // Any - как отсутствие ограничения
                    }
                    var externalReferencesRange = constants.ExternalReferencesRange;
                    if (externalReferencesRange.HasValue &&
                    ↪  externalReferencesRange.Value.Contains(value))
                    {
                        return Add(ExternalSourcesTreeMethods.CountUsages(value),
                        ↪  ExternalTargetsTreeMethods.CountUsages(value));
                    }
                    else
                    {
                        if (_useLinkedList)
                        {
                            return Add(InternalSourcesListMethods.CountUsages(value),
                            ↪  InternalTargetsTreeMethods.CountUsages(value));
                        }
                        else
                        {
                            return Add(InternalSourcesTreeMethods.CountUsages(value),
                            ↪  InternalTargetsTreeMethods.CountUsages(value));
                        }
                    }
                }
                else
                {
                    if (!Exists(index))
                    {
                        return GetZero();
                    }
                    if (AreEqual(value, any))
                    {
                        return GetOne();
                    }
                    ref var storedLinkValue = ref GetLinkDataPartReference(index);
                    if (AreEqual(storedLinkValue.Source, value) ||
                    ↪  AreEqual(storedLinkValue.Target, value))
                    {
                        return GetOne();
                    }
                    return GetZero();
                }
            }
            if (restriction.Count == 3)
            {
                var externalReferencesRange = constants.ExternalReferencesRange;
                var source = this.GetSource(restriction);
                var target = this.GetTarget(restriction);
                if (AreEqual(index, any))
                {
                    if (AreEqual(source, any) && AreEqual(target, any))
                    {
                        return Total;
                    }
                    else if (AreEqual(source, any))
                    {
                        if (externalReferencesRange.HasValue &&
                        ↪  externalReferencesRange.Value.Contains(target))
                        {
                            return ExternalTargetsTreeMethods.CountUsages(target);
                        }
                        else
                        {
                            return InternalTargetsTreeMethods.CountUsages(target);
                        }
                    }
                    else if (AreEqual(target, any))
                    {
                        if (externalReferencesRange.HasValue &&
                        ↪  externalReferencesRange.Value.Contains(source))
                        {
                            return ExternalSourcesTreeMethods.CountUsages(source);
                        }
                        else
                        {
                            if (_useLinkedList)
                            {
```

```
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);
466                         }
467                     }
468                     return AreEqual(link, constants.Null) ? GetZero() : GetOne();
469                 }
470             }
471             else
472             {
473                 if (!Exists(index))
474                 {
475                     return GetZero();
476                 }
477                 if (AreEqual(source, any) && AreEqual(target, any))
478                 {
479                     return GetOne();
480                 }
481                 ref var storedLinkValue = ref GetLinkDataPartReference(index);
482                 if (!AreEqual(source, any) && !AreEqual(target, any))
483                 {
```

```
                        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))
                {
                    if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
                    {
                        return @break;
                    }
                }
                return @break;
            }
            var @continue = constants.Continue;
            var any = constants.Any;
            var index = this.GetIndex(restriction);
            if (restriction.Count == 1)
            {
                if (AreEqual(index, any))
                {
                    return Each(Array.Empty<TLinkAddress>(), handler);
                }
                if (!Exists(index))
```

```
                {
                    return @continue;
                }
                return handler(GetLinkStruct(index));
            }
            if (restriction.Count == 2)
            {
                var value = restriction[1];
                if (AreEqual(index, any))
                {
                    if (AreEqual(value, any))
                    {
                        return Each(Array.Empty<TLinkAddress>(), handler);
                    }
                    if (AreEqual(Each(new Link<TLinkAddress>(index, value, any), handler),
                    ↪  @break))
                    {
                        return @break;
                    }
                    return Each(new Link<TLinkAddress>(index, any, value), handler);
                }
                else
                {
                    if (!Exists(index))
                    {
                        return @continue;
                    }
                    if (AreEqual(value, any))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    ref var storedLinkValue = ref GetLinkDataPartReference(index);
                    if (AreEqual(storedLinkValue.Source, value) ||
                        AreEqual(storedLinkValue.Target, value))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    return @continue;
                }
            }
            if (restriction.Count == 3)
            {
                var externalReferencesRange = constants.ExternalReferencesRange;
                var source = this.GetSource(restriction);
                var target = this.GetTarget(restriction);
                if (AreEqual(index, any))
                {
                    if (AreEqual(source, any) && AreEqual(target, any))
                    {
                        return Each(Array.Empty<TLinkAddress>(), handler);
                    }
                    else if (AreEqual(source, any))
                    {
                        if (externalReferencesRange.HasValue &&
                        ↪  externalReferencesRange.Value.Contains(target))
                        {
                            return ExternalTargetsTreeMethods.EachUsage(target, handler);
                        }
                        else
                        {
                            return InternalTargetsTreeMethods.EachUsage(target, handler);
                        }
                    }
                    else if (AreEqual(target, any))
                    {
                        if (externalReferencesRange.HasValue &&
                        ↪  externalReferencesRange.Value.Contains(source))
                        {
                            return ExternalSourcesTreeMethods.EachUsage(source, handler);
                        }
                        else
                        {
                            if (_useLinkedList)
                            {
                                return InternalSourcesListMethods.EachUsage(source, handler);
                            }
                            else
                            {
```

```
                            return InternalSourcesTreeMethods.EachUsage(source, handler);
                        }
                    }
                }
                else //if(source != Any && target != Any)
                {
                    TLinkAddress link;
                    if (externalReferencesRange.HasValue)
                    {
                        if (externalReferencesRange.Value.Contains(source) &&
                        ↪  externalReferencesRange.Value.Contains(target))
                        {
                            link = ExternalSourcesTreeMethods.Search(source, target);
                        }
                        else if (externalReferencesRange.Value.Contains(source))
                        {
                            link = InternalTargetsTreeMethods.Search(source, target);
                        }
                        else if (externalReferencesRange.Value.Contains(target))
                        {
                            if (_useLinkedList)
                            {
                                link = ExternalSourcesTreeMethods.Search(source, target);
                            }
                            else
                            {
                                link = InternalSourcesTreeMethods.Search(source, target);
                            }
                        }
                        else
                        {
                            if (_useLinkedList ||
                            ↪  GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
                            ↪  InternalTargetsTreeMethods.CountUsages(target)))
                            {
                                link = InternalTargetsTreeMethods.Search(source, target);
                            }
                            else
                            {
                                link = InternalSourcesTreeMethods.Search(source, target);
                            }
                        }
                    }
                    else
                    {
                        if (_useLinkedList ||
                        ↪  GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
                        ↪  InternalTargetsTreeMethods.CountUsages(target)))
                        {
                            link = InternalTargetsTreeMethods.Search(source, target);
                        }
                        else
                        {
                            link = InternalSourcesTreeMethods.Search(source, target);
                        }
                    }
                    return AreEqual(link, constants.Null) ? @continue :
                    ↪  handler(GetLinkStruct(link));
                }
            }
            else
            {
                if (!Exists(index))
                {
                    return @continue;
                }
                if (AreEqual(source, any) && AreEqual(target, any))
                {
                    return handler(GetLinkStruct(index));
                }
                ref var storedLinkValue = ref GetLinkDataPartReference(index);
                if (!AreEqual(source, any) && !AreEqual(target, any))
                {
                    if (AreEqual(storedLinkValue.Source, source) &&
                        AreEqual(storedLinkValue.Target, target))
                    {
                        return handler(GetLinkStruct(index));
                    }
```

```
                        return @continue;
                    }
                    var value = default(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 = this.GetIndex(restriction);
            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);
                    }
                }
            }
            if (!AreEqual(target, @null))
            {
                if (externalReferencesRange.HasValue &&
                ↪  externalReferencesRange.Value.Contains(target))
                {
                    ExternalTargetsTreeMethods.Detach(ref rootAsTarget, linkIndex);
                }
                else
                {
                    InternalTargetsTreeMethods.Detach(ref
                    ↪  GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
                }
            }
            source = link.Source = this.GetSource(substitution);
```

```
774             target = link.Target = this.GetTarget(substitution);
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 != null ? handler(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)))
827                 {
828                     _dataMemory.ReservedCapacity += _dataMemoryReservationStepInBytes;
829                     _indexMemory.ReservedCapacity += _indexMemoryReservationStepInBytes;
830                     SetPointers(_dataMemory, _indexMemory);
831                     header = ref GetHeaderReference();
832                     header.ReservedLinks = ConvertToAddress(_dataMemory.ReservedCapacity /
                            ↪ LinkDataPartSizeInBytes);
833                 }
834                 freeLink = header.AllocatedLinks = Increment(header.AllocatedLinks);
835                 _dataMemory.UsedCapacity += LinkDataPartSizeInBytes;
836                 _indexMemory.UsedCapacity += LinkIndexPartSizeInBytes;
837             }
838             return handler != null ? handler(null, GetLinkStruct(freeLink)) : Constants.Continue;
839         }
840 
841         /// <summary>
842         /// <para>
```

```
843             /// Deletes the substitution.
844             /// </para>
845             /// <para></para>
846             /// </summary>
847             /// <param name="restriction">
848             /// <para>The substitution.</para>
849             /// <para></para>
850             /// </param>
851             [MethodImpl(MethodImplOptions.AggressiveInlining)]
852             public virtual TLinkAddress Delete(IList<TLinkAddress>? restriction,
                ↪ WriteHandler<TLinkAddress>? handler)
853             {
854                 ref var header = ref GetHeaderReference();
855                 var link = restriction[Constants.IndexPart];
856                 var before = GetLinkStruct(link);
857                 if (LessThan(link, header.AllocatedLinks))
858                 {
859                     UnusedLinksListMethods.AttachAsFirst(link);
860                 }
861                 else if (AreEqual(link, header.AllocatedLinks))
862                 {
863                     header.AllocatedLinks = Decrement(header.AllocatedLinks);
864                     _dataMemory.UsedCapacity -= LinkDataPartSizeInBytes;
865                     _indexMemory.UsedCapacity -= LinkIndexPartSizeInBytes;
866                     // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
                        ↪ пока не дойдём до первой существующей связи
867                     // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
868                     while (GreaterThan(header.AllocatedLinks, GetZero()) &&
                        ↪ IsUnusedLink(header.AllocatedLinks))
869                     {
870                         UnusedLinksListMethods.Detach(header.AllocatedLinks);
871                         header.AllocatedLinks = Decrement(header.AllocatedLinks);
872                         _dataMemory.UsedCapacity -= LinkDataPartSizeInBytes;
873                         _indexMemory.UsedCapacity -= LinkIndexPartSizeInBytes;
874                     }
875                 }
876                 return handler != null ? handler(before, null) : Constants.Continue;
877             }
878
879             /// <summary>
880             /// <para>
881             /// Gets the link struct using the specified link index.
882             /// </para>
883             /// <para></para>
884             /// </summary>
885             /// <param name="linkIndex">
886             /// <para>The link index.</para>
887             /// <para></para>
888             /// </param>
889             /// <returns>
890             /// <para>A list of t link</para>
891             /// <para></para>
892             /// </returns>
893             [MethodImpl(MethodImplOptions.AggressiveInlining)]
894             public IList<TLinkAddress>? GetLinkStruct(TLinkAddress linkIndex)
895             {
896                 ref var link = ref GetLinkDataPartReference(linkIndex);
897                 return new Link<TLinkAddress>(linkIndex, link.Source, link.Target);
898             }
899
900             /// <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>
```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual void ResetPointers()
        {
            InternalSourcesListMethods = null;
            InternalSourcesTreeMethods = null;
            ExternalSourcesTreeMethods = null;
            InternalTargetsTreeMethods = null;
            ExternalTargetsTreeMethods = null;
            UnusedLinksListMethods = 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 abstract ref LinksHeader<TLinkAddress> GetHeaderReference();

        /// <summary>
        /// <para>
        /// Gets the link data part 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 data part of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract ref RawLinkDataPart<TLinkAddress>
        ↪  GetLinkDataPartReference(TLinkAddress linkIndex);

        /// <summary>
        /// <para>
        /// Gets the link index part 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 index part of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract ref RawLinkIndexPart<TLinkAddress>
        ↪  GetLinkIndexPartReference(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)
```

```
992             && !IsUnusedLink(link);
993 
994         /// <summary>
995         /// <para>
996         /// Determines whether this instance is unused link.
997         /// </para>
998         /// <para></para>
999         /// </summary>
1000        /// <param name="linkIndex">
1001        /// <para>The link index.</para>
1002        /// <para></para>
1003        /// </param>
1004        /// <returns>
1005        /// <para>The bool</para>
1006        /// <para></para>
1007        /// </returns>
1008        [MethodImpl(MethodImplOptions.AggressiveInlining)]
1009        protected virtual bool IsUnusedLink(TLinkAddress linkIndex)
1010        {
1011            if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
                ↪  is not needed
1012            {
1013                // TODO: Reduce access to memory in different location (should be enough to use
                    ↪  just linkIndexPart)
1014                ref var linkDataPart = ref GetLinkDataPartReference(linkIndex);
1015                ref var linkIndexPart = ref GetLinkIndexPartReference(linkIndex);
1016                return AreEqual(linkIndexPart.SizeAsTarget, default) &&
                    ↪  !AreEqual(linkDataPart.Source, default);
1017            }
1018            else
1019            {
1020                return true;
1021            }
1022        }
1023 
1024        /// <summary>
1025        /// <para>
1026        /// Gets the one.
1027        /// </para>
1028        /// <para></para>
1029        /// </summary>
1030        /// <returns>
1031        /// <para>The link</para>
1032        /// <para></para>
1033        /// </returns>
1034        [MethodImpl(MethodImplOptions.AggressiveInlining)]
1035        protected virtual TLinkAddress GetOne() => _one;
1036 
1037        /// <summary>
1038        /// <para>
1039        /// Gets the zero.
1040        /// </para>
1041        /// <para></para>
1042        /// </summary>
1043        /// <returns>
1044        /// <para>The link</para>
1045        /// <para></para>
1046        /// </returns>
1047        [MethodImpl(MethodImplOptions.AggressiveInlining)]
1048        protected virtual TLinkAddress GetZero() => default;
1049 
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">
```

```
        /// <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 GreaterOrEqualThan(TLinkAddress first, TLinkAddress second) =>
        ↪  _comparer.Compare(first, second) >= 0;

        /// <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 virtual long ConvertToInt64(TLinkAddress value) =>
        ↪  _addressToInt64Converter.Convert(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 link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLinkAddress ConvertToAddress(long value) =>
        ↪  _int64ToAddressConverter.Convert(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 virtual TLinkAddress Add(TLinkAddress first, TLinkAddress second) =>
        ↪  Arithmetic<TLinkAddress>.Add(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 virtual TLinkAddress Subtract(TLinkAddress first, TLinkAddress second) =>
        ↪  Arithmetic<TLinkAddress>.Subtract(first, second);

        /// <summary>
        /// <para>
        /// Increments the 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 virtual TLinkAddress Increment(TLinkAddress link) =>
        ↪  Arithmetic<TLinkAddress>.Increment(link);

        /// <summary>
        /// <para>
        /// Decrements the 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 virtual TLinkAddress Decrement(TLinkAddress link) =>
        ↪  Arithmetic<TLinkAddress>.Decrement(link);

        #region Disposable

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

        /// <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)
            {
                ResetPointers();
                _dataMemory.DisposeIfPossible();
                _indexMemory.DisposeIfPossible();
            }
        }

        #endregion
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Memory.Split.Generic
{
    /// <summary>
    /// <para>
    /// Represents the unused links list methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="AbsoluteCircularDoublyLinkedListMethods{TLinkAddress}"/>
    /// <seealso cref="ILinksListMethods{TLinkAddress}"/>
    public unsafe class UnusedLinksListMethods<TLinkAddress> :
    ↪  AbsoluteCircularDoublyLinkedListMethods<TLinkAddress>, ILinksListMethods<TLinkAddress>
    {
        private static readonly UncheckedConverter<TLinkAddress, long> _addressToInt64Converter
        ↪  = UncheckedConverter<TLinkAddress, long>.Default;
        private readonly byte* _links;
        private readonly byte* _header;

        /// <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);

```

```
58          /// <summary>
59          /// <para>
60          /// Gets the link data part reference using the specified link.
61          /// </para>
62          /// <para></para>
63          /// </summary>
64          /// <param name="link">
65          /// <para>The link.</para>
66          /// <para></para>
67          /// </param>
68          /// <returns>
69          /// <para>A ref raw link data part of t link</para>
70          /// <para></para>
71          /// </returns>
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          protected virtual ref RawLinkDataPart<TLinkAddress>
            ↪ GetLinkDataPartReference(TLinkAddress link) => ref
            ↪ AsRef<RawLinkDataPart<TLinkAddress>>(_links +
            ↪ (RawLinkDataPart<TLinkAddress>.SizeInBytes *
            ↪ _addressToInt64Converter.Convert(link)));
74
75          /// <summary>
76          /// <para>
77          /// Gets the first.
78          /// </para>
79          /// <para></para>
80          /// </summary>
81          /// <returns>
82          /// <para>The link</para>
83          /// <para></para>
84          /// </returns>
85          [MethodImpl(MethodImplOptions.AggressiveInlining)]
86          protected override TLinkAddress GetFirst() => GetHeaderReference().FirstFreeLink;
87
88          /// <summary>
89          /// <para>
90          /// Gets the last.
91          /// </para>
92          /// <para></para>
93          /// </summary>
94          /// <returns>
95          /// <para>The link</para>
96          /// <para></para>
97          /// </returns>
98          [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) =>
            ↪ 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 }
```

1.49 ./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  }
```

1.50 ./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>
94          /// </param>
95          /// <returns>
96          /// <para>The bool</para>
97          /// <para></para>
98          /// </returns>
99          [MethodImpl(MethodImplOptions.AggressiveInlining)]
100         public override bool Equals(object obj) => obj is RawLinkIndexPart<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(RawLinkIndexPart<TLinkAddress> other)
118             => _equalityComparer.Equals(RootAsSource, other.RootAsSource)
119             && _equalityComparer.Equals(LeftAsSource, other.LeftAsSource)
120             && _equalityComparer.Equals(RightAsSource, other.RightAsSource)
121             && _equalityComparer.Equals(SizeAsSource, other.SizeAsSource)
122             && _equalityComparer.Equals(RootAsTarget, other.RootAsTarget)
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() => (RootAsSource, LeftAsSource, RightAsSource,
            ↪ SizeAsSource, RootAsTarget, LeftAsTarget, RightAsTarget, SizeAsTarget).GetHashCode();
139 
140         [MethodImpl(MethodImplOptions.AggressiveInlining)]
141         public static bool operator ==(RawLinkIndexPart<TLinkAddress> left,
            ↪ RawLinkIndexPart<TLinkAddress> right) => left.Equals(right);
142 
143         [MethodImpl(MethodImplOptions.AggressiveInlining)]
144         public static bool operator !=(RawLinkIndexPart<TLinkAddress> left,
            ↪ RawLinkIndexPart<TLinkAddress> right) => !(left == right);
145     }
146 }
```

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

```
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 recursionless size balanced tree methods base.
12     /// </para>
13     /// <para></para>
14     /// </summary>
15     /// <seealso cref="ExternalLinksRecursionlessSizeBalancedTreeMethodsBase{TLinkAddress}"/>
16     /// <seealso cref="ILinksTreeMethods{TLinkAddress}"/>
17     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>
```

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

        /// <summary>
        /// <para>
        /// Initializes a new <see
        ↪  cref="UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase"/> 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 UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase(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>
        /// <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>
        /// <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(TLinkAddress 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(TLinkAddress first, TLinkAddress 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(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 == 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.
        /// </para>
        /// <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 }
```

1.52 ./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>
```

```
        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="UInt32ExternalLinksSizeBalancedTreeMethodsBase"/>
        ↪  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 UInt32ExternalLinksSizeBalancedTreeMethodsBase(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>
        /// <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(TLinkAddress 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(TLinkAddress first, TLinkAddress 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(TLinkAddress first, TLinkAddress second) =>
        ↪  first >= second;

```

```
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 == 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(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>
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(TLinkAddress 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>
271         /// </returns>
272         [MethodImpl(MethodImplOptions.AggressiveInlining)]
273         protected override bool LessThan(TLinkAddress first, TLinkAddress second) => first <
            ↪ second;
274
275         /// <summary>
276         /// <para>
277         /// Increments the value.
278         /// </para>
279         /// <para></para>
280         /// </summary>
281         /// <param name="value">
282         /// <para>The value.</para>
283         /// <para></para>
284         /// </param>
285         /// <returns>
286         /// <para>The link</para>
287         /// <para></para>
288         /// </returns>
289         [MethodImpl(MethodImplOptions.AggressiveInlining)]
290         protected override TLinkAddress Increment(TLinkAddress value) => ++value;
291
292         /// <summary>
293         /// <para>
294         /// Decrements the value.
295         /// </para>
296         /// <para></para>
297         /// </summary>
298         /// <param name="value">
299         /// <para>The value.</para>
300         /// <para></para>
301         /// </param>
302         /// <returns>
303         /// <para>The link</para>
304         /// <para></para>
305         /// </returns>
306         [MethodImpl(MethodImplOptions.AggressiveInlining)]
307         protected override TLinkAddress Decrement(TLinkAddress value) => --value;
308
309         /// <summary>
310         /// <para>
311         /// Adds the first.
312         /// </para>
313         /// <para></para>
314         /// </summary>
315         /// <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>
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     }
```

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

```
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 recursionless size balanced tree methods.
11      /// </para>
12      /// <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">
```

```
        /// <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 UInt32ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(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">
```

```
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>
```

```
        /// </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/UInt32ExternalLinksSourcesSizeBalancedTreeMeth

```
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 size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="UInt32ExternalLinksSizeBalancedTreeMethodsBase"/>
    public unsafe class UInt32ExternalLinksSourcesSizeBalancedTreeMethods :
    ↪  UInt32ExternalLinksSizeBalancedTreeMethodsBase
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt32ExternalLinksSourcesSizeBalancedTreeMethods"/>
        ↪  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>
```

```
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>
```

```
        /// <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;

        /// <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;
```

```
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 <
               ↪  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>
```

```
        /// <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;
        }
    }
}
```

1.55 ./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.
        /// </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 UInt32ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(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].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
        ↪  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>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [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>
```

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

        /// <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].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)
            => 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>
```

```
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 }
```

1.56 ./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>
```

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

        /// <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].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)
            => 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>
```

```
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 }
```

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

```
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 recursionless size balanced tree methods base.
12     /// </para>
13     /// <para></para>
14     /// </summary>
15     /// <seealso cref="InternalLinksRecursionlessSizeBalancedTreeMethodsBase{TLinkAddress}"/>
16     public unsafe abstract class UInt32InternalLinksRecursionlessSizeBalancedTreeMethodsBase :
       ↪ InternalLinksRecursionlessSizeBalancedTreeMethodsBase<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="UInt32InternalLinksRecursionlessSizeBalancedTreeMethodsBase"/> instance.
43         /// </para>
44         /// <para></para>
45         /// </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>
        /// <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;
```

```
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>
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.
277         /// </para>
278         /// <para></para>
279         /// </summary>
280         /// <param name="value">
281         /// <para>The value.</para>
282         /// <para></para>
283         /// </param>
284         /// <returns>
285         /// <para>The link</para>
286         /// <para></para>
287         /// </returns>
288         [MethodImpl(MethodImplOptions.AggressiveInlining)]
289         protected override TLinkAddress Increment(TLinkAddress value) => ++value;
290
291         /// <summary>
292         /// <para>
293         /// Decrements the value.
294         /// </para>
295         /// <para></para>
296         /// </summary>
297         /// <param name="value">
298         /// <para>The value.</para>
299         /// <para></para>
300         /// </param>
301         /// <returns>
302         /// <para>The link</para>
303         /// <para></para>
304         /// </returns>
305         [MethodImpl(MethodImplOptions.AggressiveInlining)]
306         protected override TLinkAddress Decrement(TLinkAddress value) => --value;
307
308         /// <summary>
309         /// <para>
310         /// Adds the first.
311         /// </para>
312         /// <para></para>
313         /// </summary>
314         /// <param name="first">
315         /// <para>The first.</para>
316         /// <para></para>
317         /// </param>
318         /// <param name="second">
319         /// <para>The second.</para>
320         /// <para></para>
321         /// </param>
322         /// <returns>
323         /// <para>The link</para>
324         /// <para></para>
325         /// </returns>
326         [MethodImpl(MethodImplOptions.AggressiveInlining)]
327         protected override TLinkAddress Add(TLinkAddress first, TLinkAddress second) => first +
            ↪ second;
328
329         /// <summary>
330         /// <para>
331         /// 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 }
```

1.58 ./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>
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 UInt32InternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLinkAddress>
            ↪  constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
            ↪  RawLinkIndexPart<TLinkAddress>* linksIndexParts, LinksHeader<TLinkAddress>* header)
64             : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
65         {
```

```
            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>
        /// <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(TLinkAddress value) => value > 0U;

        /// <summary>
        /// <para>
        /// Determines whether this instance greater than.
        /// </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 == 0UL; //
            ↪ value is always >= 0 for ulong
214
215         /// <summary>
216         /// <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.
        /// </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);
    }
}
```

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

```
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>
```

```
14      /// <seealso cref="InternalLinksSourcesLinkedListMethods{TLinkAddress}"/>
15      public unsafe class UInt32InternalLinksSourcesLinkedListMethods :
        ↪  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="UInt32InternalLinksSourcesLinkedListMethods"/> 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 UInt32InternalLinksSourcesLinkedListMethods(LinksConstants<TLinkAddress>
            ↪  constants, RawLinkDataPart<TLinkAddress>* linksDataParts,
            ↪  RawLinkIndexPart<TLinkAddress>* linksIndexParts)
40              : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts)
41          {
42              _linksDataParts = linksDataParts;
43              _linksIndexParts = linksIndexParts;
44          }
45
46          /// <summary>
47          /// <para>
48          /// Gets the link data part reference using the specified link.
49          /// </para>
50          /// <para></para>
51          /// </summary>
52          /// <param name="link">
53          /// <para>The link.</para>
54          /// <para></para>
55          /// </param>
56          /// <returns>
57          /// <para>A ref raw link data part of t link</para>
58          /// <para></para>
59          /// </returns>
60          [MethodImpl(MethodImplOptions.AggressiveInlining)]
61          protected override ref RawLinkDataPart<TLinkAddress>
            ↪  GetLinkDataPartReference(TLinkAddress link) => ref _linksDataParts[link];
62
63          /// <summary>
64          /// <para>
65          /// Gets the link index part reference using the specified link.
66          /// </para>
67          /// <para></para>
68          /// </summary>
69          /// <param name="link">
70          /// <para>The link.</para>
71          /// <para></para>
72          /// </param>
73          /// <returns>
74          /// <para>A ref raw link index part of t link</para>
75          /// <para></para>
76          /// </returns>
77          [MethodImpl(MethodImplOptions.AggressiveInlining)]
78          protected override ref RawLinkIndexPart<TLinkAddress>
            ↪  GetLinkIndexPartReference(TLinkAddress link) => ref _linksIndexParts[link];
79      }
80  }
```

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

```
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
```

```
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>
        /// </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;

        /// <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">
```

```
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>
```

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

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

```
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 size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="UInt32InternalLinksSizeBalancedTreeMethodsBase"/>
    public unsafe class UInt32InternalLinksSourcesSizeBalancedTreeMethods :
    ↪ UInt32InternalLinksSizeBalancedTreeMethodsBase
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt32InternalLinksSourcesSizeBalancedTreeMethods"/>
        ↪ 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 UInt32InternalLinksSourcesSizeBalancedTreeMethods(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 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>
```

```
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 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>
```

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

1.62 ./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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(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>
```

```
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     }
```

```
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 size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="UInt32InternalLinksSizeBalancedTreeMethodsBase"/>
    public unsafe class UInt32InternalLinksTargetsSizeBalancedTreeMethods :
    ↪ UInt32InternalLinksSizeBalancedTreeMethodsBase
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt32InternalLinksTargetsSizeBalancedTreeMethods"/>
        ↪ 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 UInt32InternalLinksTargetsSizeBalancedTreeMethods(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 link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLinkAddress GetLeftReference(TLinkAddress 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 link</para>
        /// <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>
```

```
        /// <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].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);
    }
}
```

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

```
using System;
using System.Runtime.CompilerServices;
using Platform.Singletons;
using Platform.Memory;
using Platform.Data.Doublets.Memory.Split.Generic;
using 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 split memory links.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="SplitMemoryLinksBase{TLinkAddress}"/>
    public unsafe class UInt32SplitMemoryLinks : SplitMemoryLinksBase<TLinkAddress>
    {
        private readonly Func<ILinksTreeMethods<TLinkAddress>> _createInternalSourceTreeMethods;
        private readonly Func<ILinksTreeMethods<TLinkAddress>> _createExternalSourceTreeMethods;
        private readonly Func<ILinksTreeMethods<TLinkAddress>> _createInternalTargetTreeMethods;
        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>
```

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

        /// <summary>
        /// <para>
        /// Resets the pointers.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ResetPointers()
        {
            base.ResetPointers();
            _linksDataParts = null;
            _linksIndexParts = 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 *_header;

        /// <summary>
        /// <para>
        /// Gets the link data part 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 data part of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkDataPart<TLinkAddress>
        ↪  GetLinkDataPartReference(TLinkAddress linkIndex) => ref _linksDataParts[linkIndex];

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

```
        /// </param>
        /// <returns>
        /// <para>The long</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override long ConvertToInt64(TLinkAddress value) => 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 link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress ConvertToAddress(long value) => (TLinkAddress)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>
        /// Increments the 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 Increment(TLinkAddress link) => ++link;

        /// <summary>
        /// <para>
        /// Decrements the 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 Decrement(TLinkAddress link) => --link;
    }
}
```

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

```
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 unused links list methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="UnusedLinksListMethods{TLinkAddress}"/>
    public unsafe class UInt32UnusedLinksListMethods : UnusedLinksListMethods<TLinkAddress>
    {
        private readonly RawLinkDataPart<TLinkAddress>* _links;
        private readonly LinksHeader<TLinkAddress>* _header;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt32UnusedLinksListMethods"/> 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 UInt32UnusedLinksListMethods(RawLinkDataPart<TLinkAddress>* links,
        ↪  LinksHeader<TLinkAddress>* 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;
    }
}
```

1.66 ./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;

        /// <summary>
        /// <para>
        /// Initializes a new <see
        ↪ cref="UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase"/> 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 UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase(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 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)]
        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>
```

```
280         /// </summary>
281         /// <param name="value">
282         /// <para>The value.</para>
283         /// <para></para>
284         /// </param>
285         /// <returns>
286         /// <para>The ulong</para>
287         /// <para></para>
288         /// </returns>
289         [MethodImpl(MethodImplOptions.AggressiveInlining)]
290         protected override ulong Increment(ulong value) => ++value;
291
292         /// <summary>
293         /// <para>
294         /// Decrements the value.
295         /// </para>
296         /// <para></para>
297         /// </summary>
298         /// <param name="value">
299         /// <para>The value.</para>
300         /// <para></para>
301         /// </param>
302         /// <returns>
303         /// <para>The ulong</para>
304         /// <para></para>
305         /// </returns>
306         [MethodImpl(MethodImplOptions.AggressiveInlining)]
307         protected override ulong Decrement(ulong value) => --value;
308
309         /// <summary>
310         /// <para>
311         /// Adds the first.
312         /// </para>
313         /// <para></para>
314         /// </summary>
315         /// <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 ulong</para>
325         /// <para></para>
326         /// </returns>
327         [MethodImpl(MethodImplOptions.AggressiveInlining)]
328         protected override ulong Add(ulong first, ulong 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 ulong</para>
346         /// <para></para>
347         /// </returns>
348         [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>
```

```
        /// <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];
            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>
```

```
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/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>
43          /// Initializes a new <see cref="UInt64ExternalLinksSizeBalancedTreeMethodsBase"/>
                ↪ 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>
```

```
        /// <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 > 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>
```

```
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>
271         /// </returns>
272         [MethodImpl(MethodImplOptions.AggressiveInlining)]
273         protected override bool LessThan(ulong first, ulong second) => first < second;
274
275         /// <summary>
276         /// <para>
277         /// Increments the value.
278         /// </para>
279         /// <para></para>
280         /// </summary>
281         /// <param name="value">
282         /// <para>The value.</para>
283         /// <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)]
```

```
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>
```

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

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

```
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 sources recursionless size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase"/>
    public unsafe class UInt64ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods :
    ↪ UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see
        ↪ cref="UInt64ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods"/> 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 UInt64ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(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>
```

```
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>
```

```
        /// <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>
```

```
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
268             /// <summary>
269             /// <para>
270             /// Clears the node using the specified node.
271             /// </para>
272             /// <para></para>
273             /// </summary>
274             /// <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/UInt64ExternalLinksSourcesSizeBalancedTreeMeth

```
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 sources size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="UInt64ExternalLinksSizeBalancedTreeMethodsBase"/>
    public unsafe class UInt64ExternalLinksSourcesSizeBalancedTreeMethods :
    ↪ UInt64ExternalLinksSizeBalancedTreeMethodsBase
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt64ExternalLinksSourcesSizeBalancedTreeMethods"/>
        ↪ 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 UInt64ExternalLinksSourcesSizeBalancedTreeMethods(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 link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
        ↪ LinksIndexParts[node].LeftAsSource;

```

```
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         {
```

```
            ref var link = ref LinksIndexParts[node];
            link.LeftAsSource = Zero;
            link.RightAsSource = Zero;
            link.SizeAsSource = Zero;
        }
    }
}
```

1.70 ./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>
        /// <para>The ref link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLinkAddress GetLeftReference(TLinkAddress node) => ref
        ↪ LinksIndexParts[node].LeftAsTarget;

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

```
        /// <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">
        /// <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].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) =>
        ↪ LinksIndexParts[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() => Header->RootAsTarget;

        /// <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].Target;

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

```
}
```

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

```
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 size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="UInt64ExternalLinksSizeBalancedTreeMethodsBase"/>
    public unsafe class UInt64ExternalLinksTargetsSizeBalancedTreeMethods :
    ↪  UInt64ExternalLinksSizeBalancedTreeMethodsBase
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt64ExternalLinksTargetsSizeBalancedTreeMethods"/>
        ↪  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 UInt64ExternalLinksTargetsSizeBalancedTreeMethods(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 link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLinkAddress GetLeftReference(TLinkAddress 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>
```

```
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>
```

```
        /// <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)
            => 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(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;
        }
    }
}
```

1.72 ./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)]
        protected UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase(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 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;

        /// <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>
```

```
232         /// </returns>
233         [MethodImpl(MethodImplOptions.AggressiveInlining)]
234         protected override bool LessOrEqualThan(ulong first, ulong 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(ulong 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(ulong first, ulong second) => first < second;
273 
274         /// <summary>
275         /// <para>
276         /// Increments the value.
277         /// </para>
278         /// <para></para>
279         /// </summary>
280         /// <param name="value">
281         /// <para>The value.</para>
282         /// <para></para>
283         /// </param>
284         /// <returns>
285         /// <para>The ulong</para>
286         /// <para></para>
287         /// </returns>
288         [MethodImpl(MethodImplOptions.AggressiveInlining)]
289         protected override ulong Increment(ulong value) => ++value;
290 
291         /// <summary>
292         /// <para>
293         /// Decrements the value.
294         /// </para>
295         /// <para></para>
296         /// </summary>
297         /// <param name="value">
298         /// <para>The value.</para>
299         /// <para></para>
300         /// </param>
301         /// <returns>
302         /// <para>The ulong</para>
303         /// <para></para>
304         /// </returns>
305         [MethodImpl(MethodImplOptions.AggressiveInlining)]
306         protected override ulong Decrement(ulong value) => --value;
307 
308         /// <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);
    }
}
```

1.73 ./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}"/>
    public unsafe abstract class UInt64InternalLinksSizeBalancedTreeMethodsBase :
    ↪  InternalLinksSizeBalancedTreeMethodsBase<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="UInt64InternalLinksSizeBalancedTreeMethodsBase"/>
        ↪  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 UInt64InternalLinksSizeBalancedTreeMethodsBase(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 > 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>
```

```
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(ulong 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(ulong value) => value == 0UL; // 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(ulong first, ulong 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(ulong 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">
```

```
        /// <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>
```

```
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 }
```

1.74 ./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         {
42             _linksDataParts = linksDataParts;
43             _linksIndexParts = linksIndexParts;
44         }
45
46         /// <summary>
47         /// <para>
48         /// Gets the link data part reference using the specified link.
49         /// </para>
50         /// <para></para>
51         /// </summary>
52         /// <param name="link">
53         /// <para>The link.</para>
54         /// <para></para>
55         /// </param>
56         /// <returns>
57         /// <para>A ref raw link data part of t link</para>
58         /// <para></para>
59         /// </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];
    }
}
```

1.75 ./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">
```

```
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 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     }
```

```
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 size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="UInt64InternalLinksSizeBalancedTreeMethodsBase"/>
    public unsafe class UInt64InternalLinksSourcesSizeBalancedTreeMethods :
    ↪ UInt64InternalLinksSizeBalancedTreeMethodsBase
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt64InternalLinksSourcesSizeBalancedTreeMethods"/>
        ↪ 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 UInt64InternalLinksSourcesSizeBalancedTreeMethods(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 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>
```

```
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 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>
```

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

1.77 ./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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    /// <summary>
    /// <para>
    /// Represents the int 64 internal links targets recursionless size balanced tree methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase"/>
    public unsafe class UInt64InternalLinksTargetsRecursionlessSizeBalancedTreeMethods :
    ↪ UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see
        ↪ cref="UInt64InternalLinksTargetsRecursionlessSizeBalancedTreeMethods"/> instance.
        /// </para>
        /// <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>
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">
```

```
        /// <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].SizeAsTarget = 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].RootAsTarget;

        /// <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].Target;

        /// <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].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);
    }
}
```

1.78 ./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>
        /// <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>
```

```
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 }
```

1.79 ./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;
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>
```

```
        [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;
```

```
355
356         /// <summary>
357         /// <para>
358         /// Gets the one.
359         /// </para>
360         /// <para></para>
361         /// </summary>
362         /// <returns>
363         /// <para>The ulong</para>
364         /// <para></para>
365         /// </returns>
366         [MethodImpl(MethodImplOptions.AggressiveInlining)]
367         protected override ulong GetOne() => 1UL;
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(ulong value) => (long)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 ulong</para>
398         /// <para></para>
399         /// </returns>
400         [MethodImpl(MethodImplOptions.AggressiveInlining)]
401         protected override ulong ConvertToAddress(long value) => (ulong)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 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     }
```

1.80 ./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>
11      /// Represents the int 64 unused links list methods.
12      /// </para>
13      /// <para></para>
14      /// </summary>
15      /// <seealso cref="UnusedLinksListMethods{TLinkAddress}"/>
16      public unsafe class UInt64UnusedLinksListMethods : UnusedLinksListMethods<TLinkAddress>
17      {
18          private readonly RawLinkDataPart<ulong>* _links;
19          private readonly LinksHeader<ulong>* _header;
20
21          /// <summary>
22          /// <para>
23          /// Initializes a new <see cref="UInt64UnusedLinksListMethods"/> instance.
24          /// </para>
25          /// <para></para>
26          /// </summary>
27          /// <param name="links">
28          /// <para>A links.</para>
29          /// <para></para>
30          /// </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;
    }
}
```

1.81 ./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

namespace Platform.Data.Doublets.Memory.United.Generic
{
    /// <summary>
    /// <para>
    /// Represents the links avl balanced tree methods base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="SizedAndThreadedAVLBalancedTreeMethods{TLinkAddress}"/>
    /// <seealso cref="ILinksTreeMethods{TLinkAddress}"/>
    public unsafe abstract class LinksAvlBalancedTreeMethodsBase<TLinkAddress> :
    ↪  SizedAndThreadedAVLBalancedTreeMethods<TLinkAddress>, ILinksTreeMethods<TLinkAddress>
    {
        private static readonly UncheckedConverter<TLinkAddress, long> _addressToInt64Converter
        ↪  = UncheckedConverter<TLinkAddress, long>.Default;
        private static readonly UncheckedConverter<TLinkAddress, int> _addressToInt32Converter =
        ↪  UncheckedConverter<TLinkAddress, int>.Default;
        private static readonly UncheckedConverter<bool, TLinkAddress> _boolToAddressConverter =
        ↪  UncheckedConverter<bool, TLinkAddress>.Default;
        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;
85          }
86
87          /// <summary>
88          /// <para>
89          /// Gets the tree root.
90          /// </para>
91          /// <para></para>
92          /// </summary>
93          /// <returns>
94          /// <para>The link</para>
95          /// <para></para>
96          /// </returns>
97          [MethodImpl(MethodImplOptions.AggressiveInlining)]
98          protected abstract TLinkAddress GetTreeRoot();
99
100         /// <summary>
101         /// <para>
102         /// Gets the base part value using the specified link.
103         /// </para>
104         /// <para></para>
```

```
105         /// </summary>
106         /// <param name="link">
107         /// <para>The link.</para>
108         /// <para></para>
109         /// </param>
110         /// <returns>
111         /// <para>The link</para>
112         /// <para></para>
113         /// </returns>
114         [MethodImpl(MethodImplOptions.AggressiveInlining)]
115         protected abstract TLinkAddress GetBasePartValue(TLinkAddress link);
116 
117         /// <summary>
118         /// <para>
119         /// Determines whether this instance first is to the right of second.
120         /// </para>
121         /// <para></para>
122         /// </summary>
123         /// <param name="source">
124         /// <para>The source.</para>
125         /// <para></para>
126         /// </param>
127         /// <param name="target">
128         /// <para>The target.</para>
129         /// <para></para>
130         /// </param>
131         /// <param name="rootSource">
132         /// <para>The root source.</para>
133         /// <para></para>
134         /// </param>
135         /// <param name="rootTarget">
136         /// <para>The root target.</para>
137         /// <para></para>
138         /// </param>
139         /// <returns>
140         /// <para>The bool</para>
141         /// <para></para>
142         /// </returns>
143         [MethodImpl(MethodImplOptions.AggressiveInlining)]
144         protected abstract bool FirstIsToTheRightOfSecond(TLinkAddress source, TLinkAddress
            ↪  target, TLinkAddress rootSource, TLinkAddress rootTarget);
145 
146         /// <summary>
147         /// <para>
148         /// Determines whether this instance first is to the left of second.
149         /// </para>
150         /// <para></para>
151         /// </summary>
152         /// <param name="source">
153         /// <para>The source.</para>
154         /// <para></para>
155         /// </param>
156         /// <param name="target">
157         /// <para>The target.</para>
158         /// <para></para>
159         /// </param>
160         /// <param name="rootSource">
161         /// <para>The root source.</para>
162         /// <para></para>
163         /// </param>
164         /// <param name="rootTarget">
165         /// <para>The root target.</para>
166         /// <para></para>
167         /// </param>
168         /// <returns>
169         /// <para>The bool</para>
170         /// <para></para>
171         /// </returns>
172         [MethodImpl(MethodImplOptions.AggressiveInlining)]
173         protected abstract bool FirstIsToTheLeftOfSecond(TLinkAddress source, TLinkAddress
            ↪  target, TLinkAddress rootSource, TLinkAddress rootTarget);
174 
175         /// <summary>
176         /// <para>
177         /// Gets the header reference.
178         /// </para>
179         /// <para></para>
180         /// </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.
```

```
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>
309             [MethodImpl(MethodImplOptions.AggressiveInlining)]
310             protected virtual void SetSizeValue(ref TLinkAddress storedValue, TLinkAddress size) =>
                ↪  storedValue = Bit<TLinkAddress>.PartialWrite(storedValue, size, 5, -5);
311
312             /// <summary>
313             /// <para>
314             /// Determines whether this instance get left is child value.
315             /// </para>
316             /// <para></para>
317             /// </summary>
318             /// <param name="value">
319             /// <para>The value.</para>
320             /// <para></para>
321             /// </param>
322             /// <returns>
323             /// <para>The bool</para>
324             /// <para></para>
325             /// </returns>
326             [MethodImpl(MethodImplOptions.AggressiveInlining)]
327             protected virtual bool GetLeftIsChildValue(TLinkAddress value)
```

```
328         {
329             unchecked
330             {
331                 return _addressToBoolConverter.Convert(Bit<TLinkAddress>.PartialRead(value, 4,
                        ↪  1));
332                 //return !EqualityComparer.Equals(Bit<TLinkAddress>.PartialRead(value, 4, 1),
                        ↪  default);
333             }
334         }
335 
336         /// <summary>
337         /// <para>
338         /// Sets the left is child value using the specified stored value.
339         /// </para>
340         /// <para></para>
341         /// </summary>
342         /// <param name="storedValue">
343         /// <para>The stored value.</para>
344         /// <para></para>
345         /// </param>
346         /// <param name="value">
347         /// <para>The value.</para>
348         /// <para></para>
349         /// </param>
350         [MethodImpl(MethodImplOptions.AggressiveInlining)]
351         protected virtual void SetLeftIsChildValue(ref TLinkAddress storedValue, bool value)
352         {
353             unchecked
354             {
355                 var previousValue = storedValue;
356                 var modified = Bit<TLinkAddress>.PartialWrite(previousValue,
                        ↪  _boolToAddressConverter.Convert(value), 4, 1);
357                 storedValue = modified;
358             }
359         }
360 
361         /// <summary>
362         /// <para>
363         /// Determines whether this instance get right is child value.
364         /// </para>
365         /// <para></para>
366         /// </summary>
367         /// <param name="value">
368         /// <para>The value.</para>
369         /// <para></para>
370         /// </param>
371         /// <returns>
372         /// <para>The bool</para>
373         /// <para></para>
374         /// </returns>
375         [MethodImpl(MethodImplOptions.AggressiveInlining)]
376         protected virtual bool GetRightIsChildValue(TLinkAddress value)
377         {
378             unchecked
379             {
380                 return _addressToBoolConverter.Convert(Bit<TLinkAddress>.PartialRead(value, 3,
                        ↪  1));
381                 //return !EqualityComparer.Equals(Bit<TLinkAddress>.PartialRead(value, 3, 1),
                        ↪  default);
382             }
383         }
384 
385         /// <summary>
386         /// <para>
387         /// Sets the right is child value using the specified stored value.
388         /// </para>
389         /// <para></para>
390         /// </summary>
391         /// <param name="storedValue">
392         /// <para>The stored value.</para>
393         /// <para></para>
394         /// </param>
395         /// <param name="value">
396         /// <para>The value.</para>
397         /// <para></para>
398         /// </param>
399         [MethodImpl(MethodImplOptions.AggressiveInlining)]
400         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
            {
                var value =
                ↪  _addressToInt32Converter.Convert(Bit<TLinkAddress>.PartialRead(storedValue,
                ↪  0, 3));
                value |= 0xF8 * ((value & 4) >> 2); // if negative, then continue ones to the
                ↪  end of sbyte
                return (sbyte)value;
            }
        }

        /// <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>
```

```
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)
```

```
        {
            var root = GetTreeRoot();
            if (EqualToZero(root))
            {
                return Continue;
            }
            TLinkAddress first = Zero, current = root;
            while (!EqualToZero(current))
            {
                var @base = GetBasePartValue(current);
                if (GreaterOrEqualThan(@base, link))
                {
                    if (AreEqual(@base, link))
                    {
                        first = current;
                    }
                    current = GetLeftOrDefault(current);
                }
                else
                {
                    current = GetRightOrDefault(current);
                }
            }
            if (!EqualToZero(first))
            {
                current = first;
                while (true)
                {
                    if (AreEqual(handler(GetLinkValues(current)), Break))
                    {
                        return Break;
                    }
                    current = GetNext(current);
                    if (EqualToZero(current) || !AreEqual(GetBasePartValue(current), link))
                    {
                        break;
                    }
                }
            }
            return Continue;
        }

        /// <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/LinksRecursionlessSizeBalancedTreeMethodsBase

```
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 recursionless size balanced tree methods base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="RecursionlessSizeBalancedTreeMethods{TLinkAddress}"/>
    /// <seealso cref="ILinksTreeMethods{TLinkAddress}"/>
    public unsafe abstract class LinksRecursionlessSizeBalancedTreeMethodsBase<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.
        /// </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="LinksRecursionlessSizeBalancedTreeMethodsBase"/>
        ↪  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 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)))
                {
                    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>
```

```
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);
360             var totalRightIgnore = Zero;
361             while (!EqualToZero(root))
362             {
363                 var @base = GetBasePartValue(root);
364                 if (LessOrEqualThan(@base, link))
365                 {
366                     root = GetRightOrDefault(root);
367                 }
368                 else
369                 {
370                     totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
371                     root = GetLeftOrDefault(root);
372                 }
373             }
374             root = GetTreeRoot();
375             var totalLeftIgnore = Zero;
376             while (!EqualToZero(root))
377             {
378                 var @base = GetBasePartValue(root);
379                 if (GreaterOrEqualThan(@base, link))
380                 {
381                     root = GetLeftOrDefault(root);
382                 }
383                 else
384                 {
```

```
                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/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;
```

```
54
55          /// <summary>
56          /// <para>
57          /// Initializes a new <see cref="LinksSizeBalancedTreeMethodsBase"/> instance.
58          /// </para>
59          /// <para></para>
60          /// </summary>
61          /// <param name="constants">
62          /// <para>A constants.</para>
63          /// <para></para>
64          /// </param>
65          /// <param name="links">
66          /// <para>A links.</para>
67          /// <para></para>
68          /// </param>
69          /// <param name="header">
70          /// <para>A header.</para>
71          /// <para></para>
72          /// </param>
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          protected LinksSizeBalancedTreeMethodsBase(LinksConstants<TLinkAddress> constants, byte*
            ↪ links, byte* header)
75          {
76              Links = links;
77              Header = header;
78              Break = constants.Break;
79              Continue = constants.Continue;
80          }
81
82          /// <summary>
83          /// <para>
84          /// Gets the tree root.
85          /// </para>
86          /// <para></para>
87          /// </summary>
88          /// <returns>
89          /// <para>The link</para>
90          /// <para></para>
91          /// </returns>
92          [MethodImpl(MethodImplOptions.AggressiveInlining)]
93          protected abstract TLinkAddress GetTreeRoot();
94
95          /// <summary>
96          /// <para>
97          /// Gets the base part value using the specified link.
98          /// </para>
99          /// <para></para>
100         /// </summary>
101         /// <param name="link">
102         /// <para>The link.</para>
103         /// <para></para>
104         /// </param>
105         /// <returns>
106         /// <para>The link</para>
107         /// <para></para>
108         /// </returns>
109         [MethodImpl(MethodImplOptions.AggressiveInlining)]
110         protected abstract TLinkAddress GetBasePartValue(TLinkAddress link);
111
112         /// <summary>
113         /// <para>
114         /// Determines whether this instance first is to the right of second.
115         /// </para>
116         /// <para></para>
117         /// </summary>
118         /// <param name="source">
119         /// <para>The source.</para>
120         /// <para></para>
121         /// </param>
122         /// <param name="target">
123         /// <para>The target.</para>
124         /// <para></para>
125         /// </param>
126         /// <param name="rootSource">
127         /// <para>The root source.</para>
128         /// <para></para>
129         /// </param>
130         /// <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.
```

```
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                 }
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);
360             var totalRightIgnore = Zero;
361             while (!EqualToZero(root))
362             {
363                 var @base = GetBasePartValue(root);
364                 if (LessOrEqualThan(@base, link))
365                 {
366                     root = GetRightOrDefault(root);
367                 }
368                 else
369                 {
370                     totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
371                     root = GetLeftOrDefault(root);
372                 }
373             }
374             root = GetTreeRoot();
375             var totalLeftIgnore = Zero;
376             while (!EqualToZero(root))
377             {
378                 var @base = GetBasePartValue(root);
379                 if (GreaterOrEqualThan(@base, link))
380                 {
381                     root = GetLeftOrDefault(root);
382                 }
383                 else
384                 {
385                     totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
386                     root = GetRightOrDefault(root);
387                 }
388             }
389             return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
390         }
391 
392         /// <summary>
393         /// <para>
394         /// Eaches the usage using the specified base.
395         /// </para>
396         /// <para></para>
397         /// </summary>
398         /// <param name="@base">
399         /// <para>The base.</para>
400         /// <para></para>
401         /// </param>
402         /// <param name="handler">
403         /// <para>The handler.</para>
404         /// <para></para>
405         /// </param>
406         /// <returns>
407         /// <para>The link</para>
408         /// <para></para>
409         /// </returns>
410         [MethodImpl(MethodImplOptions.AggressiveInlining)]
411         public TLinkAddress EachUsage(TLinkAddress @base, ReadHandler<TLinkAddress>? handler) =>
            ↪ EachUsageCore(@base, GetTreeRoot(), handler);
412 
413         // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
            ↪ low-level MSIL stack.
414         [MethodImpl(MethodImplOptions.AggressiveInlining)]
415         private TLinkAddress EachUsageCore(TLinkAddress @base, TLinkAddress link,
            ↪ ReadHandler<TLinkAddress>? handler)
416         {
417             var @continue = Continue;
418             if (EqualToZero(link))
419             {
420                 return @continue;
421             }
422             var linkBasePart = GetBasePartValue(link);
423             var @break = Break;
424             if (GreaterThan(linkBasePart, @base))
425             {
426                 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);
        }
    }
}
```

1.84 ./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;

        /// <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) =>
        ↪  GetLinkReference(node).LeftAsSource;

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

```
        /// <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(TLinkAddress node) =>
        ↪  GetBalanceValue(GetLinkReference(node).SizeAsSource);

        /// <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(TLinkAddress node, sbyte value) =>
        ↪  SetBalanceValue(ref GetLinkReference(node).SizeAsSource, value);

        /// <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().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) =>
        ↪  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>
```

```
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(firstSource, secondSource) || (AreEqual(firstSource, secondSource) &&
              ↪ LessThan(firstTarget, secondTarget));
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(firstSource, secondSource) || (AreEqual(firstSource, secondSource) &&
              ↪ GreaterThan(firstTarget, secondTarget));
362
363         /// <summary>
364         /// <para>
365         /// Clears the node using the specified node.
366         /// </para>
367         /// <para></para>
368         /// </summary>
369         /// <param name="node">
370         /// <para>The node.</para>
371         /// <para></para>
372         /// </param>
373         [MethodImpl(MethodImplOptions.AggressiveInlining)]
374         protected override void ClearNode(TLinkAddress node)
375         {
376             ref var link = ref GetLinkReference(node);
377             link.LeftAsSource = Zero;
378             link.RightAsSource = Zero;
379             link.SizeAsSource = Zero;
380         }
381     }
382 }
```

1.85 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesRecursionlessSizeBalancedTreeMetho

```
1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
```

```
5  namespace Platform.Data.Doublets.Memory.United.Generic
6  {
7      /// <summary>
8      /// <para>
9      /// Represents the links sources recursionless size balanced tree methods.
10     /// </para>
11     /// <para></para>
12     /// </summary>
13     /// <seealso cref="LinksRecursionlessSizeBalancedTreeMethodsBase{TLinkAddress}"/>
14     public unsafe class LinksSourcesRecursionlessSizeBalancedTreeMethods<TLinkAddress> :
       ↪  LinksRecursionlessSizeBalancedTreeMethodsBase<TLinkAddress>
15     {
16         /// <summary>
17         /// <para>
18         /// Initializes a new <see cref="LinksSourcesRecursionlessSizeBalancedTreeMethods"/>
           ↪  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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public LinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<TLinkAddress>
           ↪  constants, byte* links, byte* header) : base(constants, links, header) { }
36
37         /// <summary>
38         /// <para>
39         /// Gets the left reference using the specified node.
40         /// </para>
41         /// <para></para>
42         /// </summary>
43         /// <param name="node">
44         /// <para>The node.</para>
45         /// <para></para>
46         /// </param>
47         /// <returns>
48         /// <para>The ref link</para>
49         /// <para></para>
50         /// </returns>
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         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>
271         [MethodImpl(MethodImplOptions.AggressiveInlining)]
272         protected override void ClearNode(TLinkAddress node)
273         {
274             ref var link = ref GetLinkReference(node);
275             link.LeftAsSource = Zero;
276             link.RightAsSource = Zero;
277             link.SizeAsSource = Zero;
278         }
279     }
280 }
```

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

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

```
        /// <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>
        /// <returns>
        /// <para>The ref link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
        ↪  GetLinkReference(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) =>
        ↪  GetLinkReference(node).LeftAsSource;

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

```
        /// <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).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() => GetHeaderReference().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) =>
        ↪ 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;
        }
    }
}
```

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

```
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Memory.United.Generic
{
    /// <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.
        /// </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) =>
        ↪  GetLinkReference(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>
```

```
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>
```

```
        /// <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(TLinkAddress node) =>
        ↪ GetLeftIsChildValue(GetLinkReference(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(TLinkAddress node, bool value) =>
        ↪ SetLeftIsChildValue(ref GetLinkReference(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(TLinkAddress node) =>
        ↪ GetRightIsChildValue(GetLinkReference(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(TLinkAddress node, bool value) =>
        ↪ SetRightIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);

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

```
        /// <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/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>
        /// 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.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="node">
        /// <para>The node.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <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) =>
            ↪ GetLinkReference(node).SizeAsTarget;
155
```

```
        /// <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)]
```

```
230         protected override bool FirstIsToTheLeftOfSecond(TLinkAddress firstSource, TLinkAddress
            ↪ firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget) =>
            ↪ LessThan(firstTarget, secondTarget) || (AreEqual(firstTarget, secondTarget) &&
            ↪ LessThan(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(TLinkAddress firstSource, TLinkAddress
            ↪ firstTarget, TLinkAddress secondSource, TLinkAddress secondTarget) =>
            ↪ GreaterThan(firstTarget, secondTarget) || (AreEqual(firstTarget, secondTarget) &&
            ↪ GreaterThan(firstSource, secondSource));
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>
271         [MethodImpl(MethodImplOptions.AggressiveInlining)]
272         protected override void ClearNode(TLinkAddress node)
273         {
274             ref var link = ref GetLinkReference(node);
275             link.LeftAsTarget = Zero;
276             link.RightAsTarget = Zero;
277             link.SizeAsTarget = Zero;
278         }
279     }
280 }
```

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

```
1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Memory.United.Generic
6  {
7      /// <summary>
8      /// <para>
9      /// Represents the links targets size balanced tree methods.
10     /// </para>
11     /// <para></para>
12     /// </summary>
13     /// <seealso cref="LinksSizeBalancedTreeMethodsBase{TLinkAddress}"/>
14     public unsafe class LinksTargetsSizeBalancedTreeMethods<TLinkAddress> :
        ↪ LinksSizeBalancedTreeMethodsBase<TLinkAddress>
15     {
16         /// <summary>
17         /// <para>
18         /// 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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLinkAddress GetRightReference(TLinkAddress node) => ref
        ↪  GetLinkReference(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) =>
        ↪  GetLinkReference(node).LeftAsTarget;

        /// <summary>
        /// <para>
        /// Gets the right using the specified node.
        /// </para>
        /// <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) =>
            ↪ 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">
```

```
        /// <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">
```

```
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(firstTarget, secondTarget) || (AreEqual(firstTarget, secondTarget) &&
              ↪ GreaterThan(firstSource, secondSource));
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>
271         [MethodImpl(MethodImplOptions.AggressiveInlining)]
272         protected override void ClearNode(TLinkAddress node)
273         {
274             ref var link = ref GetLinkReference(node);
275             link.LeftAsTarget = Zero;
276             link.RightAsTarget = Zero;
277             link.SizeAsTarget = Zero;
278         }
279     }
280 }
```

1.90 ./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinks.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.United.Generic
10 {
11     /// <summary>
12     /// <para>
13     /// Represents the united memory links.
14     /// </para>
15     /// <para></para>
16     /// </summary>
17     /// <seealso cref="UnitedMemoryLinksBase{TLinkAddress}"/>
18     public unsafe class UnitedMemoryLinks<TLinkAddress> : UnitedMemoryLinksBase<TLinkAddress>
19     {
20         private readonly Func<ILinksTreeMethods<TLinkAddress>> _createSourceTreeMethods;
21         private readonly Func<ILinksTreeMethods<TLinkAddress>> _createTargetTreeMethods;
22         private byte* _header;
23         private byte* _links;
24
25         /// <summary>
26         /// <para>
27         /// Initializes a new <see cref="UnitedMemoryLinks"/> instance.
28         /// </para>
29         /// <para></para>
30         /// </summary>
31         /// <param name="address">
32         /// <para>A address.</para>
```

```
33          /// <para></para>
34          /// </param>
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          public UnitedMemoryLinks(string address) : this(address, DefaultLinksSizeStep) { }
37
38          /// <summary>
39          /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
            ↪  минимальным шагом расширения базы данных.
40          /// </summary>
41          /// <param name="address">Полный пусть к файлу базы данных.</param>
42          /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
            ↪  байтах.</param>
43          [MethodImpl(MethodImplOptions.AggressiveInlining)]
44          public UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
            ↪  FileMappedResizableDirectMemory(address, memoryReservationStep),
            ↪  memoryReservationStep) { }
45
46          /// <summary>
47          /// <para>
48          /// Initializes a new <see cref="UnitedMemoryLinks"/> instance.
49          /// </para>
50          /// <para></para>
51          /// </summary>
52          /// <param name="memory">
53          /// <para>A memory.</para>
54          /// <para></para>
55          /// </param>
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          public UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
            ↪  DefaultLinksSizeStep) { }
58
59          /// <summary>
60          /// <para>
61          /// Initializes a new <see cref="UnitedMemoryLinks"/> instance.
62          /// </para>
63          /// <para></para>
64          /// </summary>
65          /// <param name="memory">
66          /// <para>A memory.</para>
67          /// <para></para>
68          /// </param>
69          /// <param name="memoryReservationStep">
70          /// <para>A memory reservation step.</para>
71          /// <para></para>
72          /// </param>
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          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();
144             _links = null;
145             _header = null;
146         }
147
148         /// <summary>
149         /// <para>
150         /// Gets the header reference.
151         /// </para>
152         /// <para></para>
153         /// </summary>
154         /// <returns>
155         /// <para>A ref links header of t link</para>
156         /// <para></para>
157         /// </returns>
158         [MethodImpl(MethodImplOptions.AggressiveInlining)]
159         protected override ref LinksHeader<TLinkAddress> GetHeaderReference() => ref
            ↪ AsRef<LinksHeader<TLinkAddress>>(_header);
160
161         /// <summary>
162         /// <para>
163         /// Gets the link reference using the specified link index.
164         /// </para>
165         /// <para></para>
166         /// </summary>
167         /// <param name="linkIndex">
168         /// <para>The link index.</para>
169         /// <para></para>
170         /// </param>
171         /// <returns>
```

```
172         /// <para>A ref raw link of t link</para>
173         /// <para></para>
174         /// </returns>
175         [MethodImpl(MethodImplOptions.AggressiveInlining)]
176         protected override ref RawLink<TLinkAddress> GetLinkReference(TLinkAddress linkIndex) =>
            ↪ ref AsRef<RawLink<TLinkAddress>>(_links + (LinkSizeInBytes *
            ↪ ConvertToInt64(linkIndex)));
177     }
178 }
```

1.91 ./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinksBase.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.United.Generic
15 {
16     /// <summary>
17     /// <para>
18     /// Represents the united memory links base.
19     /// </para>
20     /// <para></para>
21     /// </summary>
22     /// <seealso cref="DisposableBase"/>
23     /// <seealso cref="ILinks{TLinkAddress}"/>
24     public abstract class UnitedMemoryLinksBase<TLinkAddress> : DisposableBase,
       ↪ ILinks<TLinkAddress>
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 LinkSizeInBytes = RawLink<TLinkAddress>.SizeInBytes;
39
40         /// <summary>
41         /// <para>
42         /// The size in bytes.
43         /// </para>
44         /// <para></para>
45         /// </summary>
46         public static readonly long LinkHeaderSizeInBytes =
           ↪ LinksHeader<TLinkAddress>.SizeInBytes;
47
48         /// <summary>
49         /// <para>
50         /// The link size in bytes.
51         /// </para>
52         /// <para></para>
53         /// </summary>
54         public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
55
56         /// <summary>
57         /// <para>
58         /// The memory.
59         /// </para>
60         /// <para></para>
61         /// </summary>
62         protected readonly IResizableDirectMemory _memory;
63         /// <summary>
```

```
64          /// <para>
65          /// The memory reservation step.
66          /// </para>
67          /// <para></para>
68          /// </summary>
69          protected readonly long _memoryReservationStep;
70
71          /// <summary>
72          /// <para>
73          /// The targets tree methods.
74          /// </para>
75          /// <para></para>
76          /// </summary>
77          protected ILinksTreeMethods<TLinkAddress> TargetsTreeMethods;
78          /// <summary>
79          /// <para>
80          /// The sources tree methods.
81          /// </para>
82          /// <para></para>
83          /// </summary>
84          protected ILinksTreeMethods<TLinkAddress> SourcesTreeMethods;
85          // 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
86          /// <summary>
87          /// <para>
88          /// The unused links list methods.
89          /// </para>
90          /// <para></para>
91          /// </summary>
92          protected ILinksListMethods<TLinkAddress> UnusedLinksListMethods;
93
94          /// <summary>
95          /// Возвращает общее число связей находящихся в хранилище.
96          /// </summary>
97          protected virtual TLinkAddress Total
98          {
99              [MethodImpl(MethodImplOptions.AggressiveInlining)]
100             get
101             {
102                 ref var header = ref GetHeaderReference();
103                 return Subtract(header.AllocatedLinks, header.FreeLinks);
104             }
105         }
106
107         /// <summary>
108         /// <para>
109         /// Gets the constants value.
110         /// </para>
111         /// <para></para>
112         /// </summary>
113         public virtual LinksConstants<TLinkAddress> Constants
114         {
115             [MethodImpl(MethodImplOptions.AggressiveInlining)]
116             get;
117         }
118
119         /// <summary>
120         /// <para>
121         /// Initializes a new <see cref="UnitedMemoryLinksBase"/> instance.
122         /// </para>
123         /// <para></para>
124         /// </summary>
125         /// <param name="memory">
126         /// <para>A memory.</para>
127         /// <para></para>
128         /// </param>
129         /// <param name="memoryReservationStep">
130         /// <para>A memory reservation step.</para>
131         /// <para></para>
132         /// </param>
133         /// <param name="constants">
134         /// <para>A constants.</para>
135         /// <para></para>
136         /// </param>
137         [MethodImpl(MethodImplOptions.AggressiveInlining)]
138         protected UnitedMemoryLinksBase(IResizableDirectMemory memory, long
            ↪  memoryReservationStep, LinksConstants<TLinkAddress> constants)
139         {
```

```
140             _memory = memory;
141             _memoryReservationStep = memoryReservationStep;
142             Constants = constants;
143         }
144 
145         /// <summary>
146         /// <para>
147         /// Initializes a new <see cref="UnitedMemoryLinksBase"/> instance.
148         /// </para>
149         /// <para></para>
150         /// </summary>
151         /// <param name="memory">
152         /// <para>A memory.</para>
153         /// <para></para>
154         /// </param>
155         /// <param name="memoryReservationStep">
156         /// <para>A memory reservation step.</para>
157         /// <para></para>
158         /// </param>
159         [MethodImpl(MethodImplOptions.AggressiveInlining)]
160         protected UnitedMemoryLinksBase(IResizableDirectMemory memory, long
            ↪ memoryReservationStep) : this(memory, memoryReservationStep,
            ↪ Default<LinksConstants<TLinkAddress>>.Instance) { }
161 
162         /// <summary>
163         /// <para>
164         /// Inits the memory.
165         /// </para>
166         /// <para></para>
167         /// </summary>
168         /// <param name="memory">
169         /// <para>The memory.</para>
170         /// <para></para>
171         /// </param>
172         /// <param name="memoryReservationStep">
173         /// <para>The memory reservation step.</para>
174         /// <para></para>
175         /// </param>
176         [MethodImpl(MethodImplOptions.AggressiveInlining)]
177         protected virtual void Init(IResizableDirectMemory memory, long memoryReservationStep)
178         {
179             if (memory.ReservedCapacity < memoryReservationStep)
180             {
181                 memory.ReservedCapacity = memoryReservationStep;
182             }
183             SetPointers(memory);
184             ref var header = ref GetHeaderReference();
185             // 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 = this.GetIndex(restriction);
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 = this.GetSource(restriction);
260                 var target = this.GetTarget(restriction);
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 = this.GetIndex(restriction);
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 = this.GetSource(restriction);
410                 var target = this.GetTarget(restriction);
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                     {
```

```
                    return handler(GetLinkStruct(index));
                }
                ref var storedLinkValue = ref GetLinkReference(index);
                if (!AreEqual(source, any) && !AreEqual(target, any))
                {
                    if (AreEqual(storedLinkValue.Source, source) &&
                        AreEqual(storedLinkValue.Target, target))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    return @continue;
                }
                var value = default(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 linkIndex = this.GetIndex(restriction);
        var before = GetLinkStruct(linkIndex);
        ref var link = ref GetLinkReference(linkIndex);
        ref var header = ref GetHeaderReference();
        ref var firstAsSource = ref header.RootAsSource;
        ref var firstAsTarget = ref header.RootAsTarget;
        // Будет корректно работать только в том случае, если пространство выделенной связи
        ↪  предварительно заполнено нулями
        if (!AreEqual(link.Source, @null))
        {
            SourcesTreeMethods.Detach(ref firstAsSource, linkIndex);
        }
        if (!AreEqual(link.Target, @null))
        {
            TargetsTreeMethods.Detach(ref firstAsTarget, linkIndex);
        }
        link.Source = this.GetSource(substitution);
        link.Target = this.GetTarget(substitution);
        if (!AreEqual(link.Source, @null))
        {
            SourcesTreeMethods.Attach(ref firstAsSource, linkIndex);
        }
        if (!AreEqual(link.Target, @null))
        {
            TargetsTreeMethods.Attach(ref firstAsTarget, linkIndex);
        }
        return handler != null ? handler(before, GetLinkStruct(linkIndex)) :
        ↪  Constants.Continue;
    }

    /// <remarks>
    /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
    ↪  пространство
    /// </remarks>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
511         public virtual TLinkAddress Create(IList<TLinkAddress>? substitution,
            ↪ WriteHandler<TLinkAddress>? handler)
512         {
513             ref var header = ref GetHeaderReference();
514             var freeLink = header.FirstFreeLink;
515             if (!AreEqual(freeLink, Constants.Null))
516             {
517                 UnusedLinksListMethods.Detach(freeLink);
518             }
519             else
520             {
521                 var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
522                 if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
523                 {
524                     throw new
                        ↪ LinksLimitReachedException<TLinkAddress>(maximumPossibleInnerReference);
525                 }
526                 if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
527                 {
528                     _memory.ReservedCapacity += _memoryReservationStep;
529                     SetPointers(_memory);
530                     header = ref GetHeaderReference();
531                     header.ReservedLinks = ConvertToAddress(_memory.ReservedCapacity /
                        ↪ LinkSizeInBytes);
532                 }
533                 freeLink = header.AllocatedLinks = Increment(header.AllocatedLinks);
534                 _memory.UsedCapacity += LinkSizeInBytes;
535             }
536             return handler != null ? handler(null, new Link<TLinkAddress>(freeLink,
                ↪ Constants.Null, Constants.Null)) : Constants.Continue;
537         }
538 
539         /// <summary>
540         /// <para>
541         /// Deletes the substitution.
542         /// </para>
543         /// <para></para>
544         /// </summary>
545         /// <param name="restriction">
546         /// <para>The substitution.</para>
547         /// <para></para>
548         /// </param>
549         [MethodImpl(MethodImplOptions.AggressiveInlining)]
550         public virtual TLinkAddress Delete(IList<TLinkAddress>? restriction,
            ↪ WriteHandler<TLinkAddress>? handler)
551         {
552             ref var header = ref GetHeaderReference();
553             var link = restriction[Constants.IndexPart];
554             var before = GetLinkStruct(link);
555             if (LessThan(link, header.AllocatedLinks))
556             {
557                 UnusedLinksListMethods.AttachAsFirst(link);
558                 // return handler?.Invoke(before, null);
559                 return handler != null ? handler(before, null) : Constants.Continue;
560             }
561             else if (AreEqual(link, header.AllocatedLinks))
562             {
563                 header.AllocatedLinks = Decrement(header.AllocatedLinks);
564                 _memory.UsedCapacity -= LinkSizeInBytes;
565                 // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
                    ↪ пока не дойдём до первой существующей связи
566                 // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
567                 while (GreaterThan(header.AllocatedLinks, GetZero()) &&
                    ↪ IsUnusedLink(header.AllocatedLinks))
568                 {
569                     UnusedLinksListMethods.Detach(header.AllocatedLinks);
570                     header.AllocatedLinks = Decrement(header.AllocatedLinks);
571                     _memory.UsedCapacity -= LinkSizeInBytes;
572                 }
573                 return handler != null ? handler(before, null) : Constants.Continue;
574             }
575             return Constants.Continue;
576         }
577 
578         /// <summary>
579         /// <para>
580         /// Gets the link struct using the specified link index.
581         /// </para>
```

```
582         /// <para></para>
583         /// </summary>
584         /// <param name="linkIndex">
585         /// <para>The link index.</para>
586         /// <para></para>
587         /// </param>
588         /// <returns>
589         /// <para>A list of t link</para>
590         /// <para></para>
591         /// </returns>
592         [MethodImpl(MethodImplOptions.AggressiveInlining)]
593         public IList<TLinkAddress>? GetLinkStruct(TLinkAddress linkIndex)
594         {
595             ref var link = ref GetLinkReference(linkIndex);
596             return new Link<TLinkAddress>(linkIndex, link.Source, link.Target);
597         }
598 
599         /// <remarks>
600         /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
              ↪  адрес реально поменялся
601         ///
602         /// Указатель this.links может быть в том же месте,
603         /// так как 0-я связь не используется и имеет такой же размер как Header,
604         /// поэтому header размещается в том же месте, что и 0-я связь
605         /// </remarks>
606         [MethodImpl(MethodImplOptions.AggressiveInlining)]
607         protected abstract void SetPointers(IResizableDirectMemory memory);
608 
609         /// <summary>
610         /// <para>
611         /// Resets the pointers.
612         /// </para>
613         /// <para></para>
614         /// </summary>
615         [MethodImpl(MethodImplOptions.AggressiveInlining)]
616         protected virtual void ResetPointers()
617         {
618             SourcesTreeMethods = null;
619             TargetsTreeMethods = null;
620             UnusedLinksListMethods = null;
621         }
622 
623         /// <summary>
624         /// <para>
625         /// Gets the header reference.
626         /// </para>
627         /// <para></para>
628         /// </summary>
629         /// <returns>
630         /// <para>A ref links header of t link</para>
631         /// <para></para>
632         /// </returns>
633         [MethodImpl(MethodImplOptions.AggressiveInlining)]
634         protected abstract ref LinksHeader<TLinkAddress> GetHeaderReference();
635 
636         /// <summary>
637         /// <para>
638         /// Gets the link reference using the specified link index.
639         /// </para>
640         /// <para></para>
641         /// </summary>
642         /// <param name="linkIndex">
643         /// <para>The link index.</para>
644         /// <para></para>
645         /// </param>
646         /// <returns>
647         /// <para>A ref raw link of t link</para>
648         /// <para></para>
649         /// </returns>
650         [MethodImpl(MethodImplOptions.AggressiveInlining)]
651         protected abstract ref RawLink<TLinkAddress> GetLinkReference(TLinkAddress linkIndex);
652 
653         /// <summary>
654         /// <para>
655         /// Determines whether this instance exists.
656         /// </para>
657         /// <para></para>
658         /// </summary>
```

```
659         /// <param name="link">
660         /// <para>The link.</para>
661         /// <para></para>
662         /// </param>
663         /// <returns>
664         /// <para>The bool</para>
665         /// <para></para>
666         /// </returns>
667         [MethodImpl(MethodImplOptions.AggressiveInlining)]
668         protected virtual bool Exists(TLinkAddress link)
669             => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
670             && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
671             && !IsUnusedLink(link);
672 
673         /// <summary>
674         /// <para>
675         /// Determines whether this instance is unused link.
676         /// </para>
677         /// <para></para>
678         /// </summary>
679         /// <param name="linkIndex">
680         /// <para>The link index.</para>
681         /// <para></para>
682         /// </param>
683         /// <returns>
684         /// <para>The bool</para>
685         /// <para></para>
686         /// </returns>
687         [MethodImpl(MethodImplOptions.AggressiveInlining)]
688         protected virtual bool IsUnusedLink(TLinkAddress linkIndex)
689         {
690             if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
                ↪  is not needed
691             {
692                 ref var link = ref GetLinkReference(linkIndex);
693                 return AreEqual(link.SizeAsSource, default) && !AreEqual(link.Source, default);
694             }
695             else
696             {
697                 return true;
698             }
699         }
700 
701         /// <summary>
702         /// <para>
703         /// Gets the one.
704         /// </para>
705         /// <para></para>
706         /// </summary>
707         /// <returns>
708         /// <para>The link</para>
709         /// <para></para>
710         /// </returns>
711         [MethodImpl(MethodImplOptions.AggressiveInlining)]
712         protected virtual TLinkAddress GetOne() => _one;
713 
714         /// <summary>
715         /// <para>
716         /// Gets the zero.
717         /// </para>
718         /// <para></para>
719         /// </summary>
720         /// <returns>
721         /// <para>The link</para>
722         /// <para></para>
723         /// </returns>
724         [MethodImpl(MethodImplOptions.AggressiveInlining)]
725         protected virtual TLinkAddress GetZero() => default;
726 
727         /// <summary>
728         /// <para>
729         /// Determines whether this instance are equal.
730         /// </para>
731         /// <para></para>
732         /// </summary>
733         /// <param name="first">
734         /// <para>The first.</para>
735         /// <para></para>
```

```
736             /// </param>
737             /// <param name="second">
738             /// <para>The second.</para>
739             /// <para></para>
740             /// </param>
741             /// <returns>
742             /// <para>The bool</para>
743             /// <para></para>
744             /// </returns>
745             [MethodImpl(MethodImplOptions.AggressiveInlining)]
746             protected virtual bool AreEqual(TLinkAddress first, TLinkAddress second) =>
                ↪ _equalityComparer.Equals(first, second);
747
748             /// <summary>
749             /// <para>
750             /// Determines whether this instance less than.
751             /// </para>
752             /// <para></para>
753             /// </summary>
754             /// <param name="first">
755             /// <para>The first.</para>
756             /// <para></para>
757             /// </param>
758             /// <param name="second">
759             /// <para>The second.</para>
760             /// <para></para>
761             /// </param>
762             /// <returns>
763             /// <para>The bool</para>
764             /// <para></para>
765             /// </returns>
766             [MethodImpl(MethodImplOptions.AggressiveInlining)]
767             protected virtual bool LessThan(TLinkAddress first, TLinkAddress second) =>
                ↪ _comparer.Compare(first, second) < 0;
768
769             /// <summary>
770             /// <para>
771             /// Determines whether this instance less or equal than.
772             /// </para>
773             /// <para></para>
774             /// </summary>
775             /// <param name="first">
776             /// <para>The first.</para>
777             /// <para></para>
778             /// </param>
779             /// <param name="second">
780             /// <para>The second.</para>
781             /// <para></para>
782             /// </param>
783             /// <returns>
784             /// <para>The bool</para>
785             /// <para></para>
786             /// </returns>
787             [MethodImpl(MethodImplOptions.AggressiveInlining)]
788             protected virtual bool LessOrEqualThan(TLinkAddress first, TLinkAddress second) =>
                ↪ _comparer.Compare(first, second) <= 0;
789
790             /// <summary>
791             /// <para>
792             /// Determines whether this instance greater than.
793             /// </para>
794             /// <para></para>
795             /// </summary>
796             /// <param name="first">
797             /// <para>The first.</para>
798             /// <para></para>
799             /// </param>
800             /// <param name="second">
801             /// <para>The second.</para>
802             /// <para></para>
803             /// </param>
804             /// <returns>
805             /// <para>The bool</para>
806             /// <para></para>
807             /// </returns>
808             [MethodImpl(MethodImplOptions.AggressiveInlining)]
809             protected virtual bool GreaterThan(TLinkAddress first, TLinkAddress second) =>
                ↪ _comparer.Compare(first, second) > 0;
```

```
810
811         /// <summary>
812         /// <para>
813         /// Determines whether this instance greater or equal than.
814         /// </para>
815         /// <para></para>
816         /// </summary>
817         /// <param name="first">
818         /// <para>The first.</para>
819         /// <para></para>
820         /// </param>
821         /// <param name="second">
822         /// <para>The second.</para>
823         /// <para></para>
824         /// </param>
825         /// <returns>
826         /// <para>The bool</para>
827         /// <para></para>
828         /// </returns>
829         [MethodImpl(MethodImplOptions.AggressiveInlining)]
830         protected virtual bool GreaterOrEqualThan(TLinkAddress first, TLinkAddress second) =>
              ↪ _comparer.Compare(first, second) >= 0;
831
832         /// <summary>
833         /// <para>
834         /// Converts the to int 64 using the specified value.
835         /// </para>
836         /// <para></para>
837         /// </summary>
838         /// <param name="value">
839         /// <para>The value.</para>
840         /// <para></para>
841         /// </param>
842         /// <returns>
843         /// <para>The long</para>
844         /// <para></para>
845         /// </returns>
846         [MethodImpl(MethodImplOptions.AggressiveInlining)]
847         protected virtual long ConvertToInt64(TLinkAddress value) =>
              ↪ _addressToInt64Converter.Convert(value);
848
849         /// <summary>
850         /// <para>
851         /// Converts the to address using the specified value.
852         /// </para>
853         /// <para></para>
854         /// </summary>
855         /// <param name="value">
856         /// <para>The value.</para>
857         /// <para></para>
858         /// </param>
859         /// <returns>
860         /// <para>The link</para>
861         /// <para></para>
862         /// </returns>
863         [MethodImpl(MethodImplOptions.AggressiveInlining)]
864         protected virtual TLinkAddress ConvertToAddress(long value) =>
              ↪ _int64ToAddressConverter.Convert(value);
865
866         /// <summary>
867         /// <para>
868         /// Adds the first.
869         /// </para>
870         /// <para></para>
871         /// </summary>
872         /// <param name="first">
873         /// <para>The first.</para>
874         /// <para></para>
875         /// </param>
876         /// <param name="second">
877         /// <para>The second.</para>
878         /// <para></para>
879         /// </param>
880         /// <returns>
881         /// <para>The link</para>
882         /// <para></para>
883         /// </returns>
884         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
885         protected virtual TLinkAddress Add(TLinkAddress first, TLinkAddress second) =>
            ↪ Arithmetic<TLinkAddress>.Add(first, second);
886
887         /// <summary>
888         /// <para>
889         /// Subtracts the first.
890         /// </para>
891         /// <para></para>
892         /// </summary>
893         /// <param name="first">
894         /// <para>The first.</para>
895         /// <para></para>
896         /// </param>
897         /// <param name="second">
898         /// <para>The second.</para>
899         /// <para></para>
900         /// </param>
901         /// <returns>
902         /// <para>The link</para>
903         /// <para></para>
904         /// </returns>
905         [MethodImpl(MethodImplOptions.AggressiveInlining)]
906         protected virtual TLinkAddress Subtract(TLinkAddress first, TLinkAddress second) =>
            ↪ Arithmetic<TLinkAddress>.Subtract(first, second);
907
908         /// <summary>
909         /// <para>
910         /// Increments the link.
911         /// </para>
912         /// <para></para>
913         /// </summary>
914         /// <param name="link">
915         /// <para>The link.</para>
916         /// <para></para>
917         /// </param>
918         /// <returns>
919         /// <para>The link</para>
920         /// <para></para>
921         /// </returns>
922         [MethodImpl(MethodImplOptions.AggressiveInlining)]
923         protected virtual TLinkAddress Increment(TLinkAddress link) =>
            ↪ Arithmetic<TLinkAddress>.Increment(link);
924
925         /// <summary>
926         /// <para>
927         /// Decrements the link.
928         /// </para>
929         /// <para></para>
930         /// </summary>
931         /// <param name="link">
932         /// <para>The link.</para>
933         /// <para></para>
934         /// </param>
935         /// <returns>
936         /// <para>The link</para>
937         /// <para></para>
938         /// </returns>
939         [MethodImpl(MethodImplOptions.AggressiveInlining)]
940         protected virtual TLinkAddress Decrement(TLinkAddress link) =>
            ↪ Arithmetic<TLinkAddress>.Decrement(link);
941
942         #region Disposable
943
944         /// <summary>
945         /// <para>
946         /// Gets the allow multiple dispose calls value.
947         /// </para>
948         /// <para></para>
949         /// </summary>
950         protected override bool AllowMultipleDisposeCalls
951         {
952             [MethodImpl(MethodImplOptions.AggressiveInlining)]
953             get => true;
954         }
955
956         /// <summary>
957         /// <para>
958         /// 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)
            {
                ResetPointers();
                _memory.DisposeIfPossible();
            }
        }

        #endregion
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Memory.United.Generic
{
    /// <summary>
    /// <para>
    /// Represents the unused links list methods.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="AbsoluteCircularDoublyLinkedListMethods{TLinkAddress}"/>
    /// <seealso cref="ILinksListMethods{TLinkAddress}"/>
    public unsafe class UnusedLinksListMethods<TLinkAddress> :
    ↪  AbsoluteCircularDoublyLinkedListMethods<TLinkAddress>, ILinksListMethods<TLinkAddress>
    {
        private static readonly UncheckedConverter<TLinkAddress, long> _addressToInt64Converter
        ↪  = UncheckedConverter<TLinkAddress, long>.Default;
        private readonly byte* _links;
        private readonly byte* _header;

        /// <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>
```

```
52          /// <para>A ref links header of t link</para>
53          /// <para></para>
54          /// </returns>
55          [MethodImpl(MethodImplOptions.AggressiveInlining)]
56          protected virtual ref LinksHeader<TLinkAddress> GetHeaderReference() => ref
            ↪  AsRef<LinksHeader<TLinkAddress>>(_header);
57
58          /// <summary>
59          /// <para>
60          /// Gets the link reference using the specified link.
61          /// </para>
62          /// <para></para>
63          /// </summary>
64          /// <param name="link">
65          /// <para>The link.</para>
66          /// <para></para>
67          /// </param>
68          /// <returns>
69          /// <para>A ref raw link of t link</para>
70          /// <para></para>
71          /// </returns>
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          protected virtual ref RawLink<TLinkAddress> GetLinkReference(TLinkAddress link) => ref
            ↪  AsRef<RawLink<TLinkAddress>>(_links + (RawLink<TLinkAddress>.SizeInBytes *
            ↪  _addressToInt64Converter.Convert(link)));
74
75          /// <summary>
76          /// <para>
77          /// Gets the first.
78          /// </para>
79          /// <para></para>
80          /// </summary>
81          /// <returns>
82          /// <para>The link</para>
83          /// <para></para>
84          /// </returns>
85          [MethodImpl(MethodImplOptions.AggressiveInlining)]
86          protected override TLinkAddress GetFirst() => GetHeaderReference().FirstFreeLink;
87
88          /// <summary>
89          /// <para>
90          /// Gets the last.
91          /// </para>
92          /// <para></para>
93          /// </summary>
94          /// <returns>
95          /// <para>The link</para>
96          /// <para></para>
97          /// </returns>
98          [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;
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) =>
                ↪ GetLinkReference(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>
```

```
        /// </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;
    }
}
```

1.93 ./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;

        /// <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(RawLink<TLinkAddress> other)
            => _equalityComparer.Equals(Source, other.Source)
            && _equalityComparer.Equals(Target, other.Target)
            && _equalityComparer.Equals(LeftAsSource, other.LeftAsSource)
            && _equalityComparer.Equals(RightAsSource, other.RightAsSource)
            && _equalityComparer.Equals(SizeAsSource, other.SizeAsSource)
            && _equalityComparer.Equals(LeftAsTarget, other.LeftAsTarget)
            && _equalityComparer.Equals(RightAsTarget, other.RightAsTarget)
            && _equalityComparer.Equals(SizeAsTarget, other.SizeAsTarget);

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

1.94 ./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.
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="UInt32LinksRecursionlessSizeBalancedTreeMethodsBase"/>
            ↪ 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 UInt32LinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<uint>
            ↪ constants, RawLink<uint>* links, LinksHeader<uint>* header)
51             : base(constants, (byte*)links, (byte*)header)
52         {
53             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 > 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;

        /// <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">
```

```
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>
```

```
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint Decrement(uint 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 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>
        /// 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(uint first, uint second)
        {
            ref var firstLink = ref Links[first];
            ref var secondLink = ref Links[second];
            return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
             ↪  secondLink.Source, secondLink.Target);
        }

```

```
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 }
```

1.95 ./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>
```

```
        /// <para>
        /// The links.
        /// </para>
        /// <para></para>
        /// </summary>
        protected new readonly RawLink<uint>* Links;
        /// <summary>
        /// <para>
        /// The header.
        /// </para>
        /// <para></para>
        /// </summary>
        protected new readonly LinksHeader<uint>* Header;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt32LinksSizeBalancedTreeMethodsBase"/> 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 UInt32LinksSizeBalancedTreeMethodsBase(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 > 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;

        /// <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>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessThan(uint first, uint 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 uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint Increment(uint value) => ++value;

        /// <summary>
        /// <para>
        /// Decrements the 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 Decrement(uint 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 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">
```

```
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 }
```

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

```
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 32 links sources recursionless size balanced tree methods.
10     /// </para>
11     /// <para></para>
12     /// </summary>
13     /// <seealso cref="UInt32LinksRecursionlessSizeBalancedTreeMethodsBase"/>
14     public unsafe class UInt32LinksSourcesRecursionlessSizeBalancedTreeMethods :
       ↪  UInt32LinksRecursionlessSizeBalancedTreeMethodsBase
15     {
16         /// <summary>
17         /// <para>
18         /// Initializes a new <see
           ↪  cref="UInt32LinksSourcesRecursionlessSizeBalancedTreeMethods"/> 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 UInt32LinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<uint>
           ↪  constants, RawLink<uint>* links, LinksHeader<uint>* 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 uint</para>
48         /// <para></para>
49         /// </returns>
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         protected override ref uint GetLeftReference(uint node) => ref Links[node].LeftAsSource;
52
53         /// <summary>
54         /// <para>
55         /// 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 = OU;
            link.RightAsSource = OU;
            link.SizeAsSource = OU;
        }
    }
}
```

```
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>
        /// </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)]
```

```
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];
276             link.LeftAsSource = 0U;
277             link.RightAsSource = 0U;
278             link.SizeAsSource = 0U;
279         }
280     }
281 }
```

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

```
1  using System.Runtime.CompilerServices;
2  
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4  
5  namespace Platform.Data.Doublets.Memory.United.Specific
6  {
7      /// <summary>
8      /// <para>
9      /// Represents the int 32 links targets recursionless size balanced tree methods.
10     /// </para>
11     /// <para></para>
12     /// </summary>
13     /// <seealso cref="UInt32LinksRecursionlessSizeBalancedTreeMethodsBase"/>
14     public unsafe class UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods :
       ↪ UInt32LinksRecursionlessSizeBalancedTreeMethodsBase
15     {
16         /// <summary>
17         /// <para>
```

```
18          /// Initializes a new <see
            ↪  cref="UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods"/> 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 UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<uint>
            ↪  constants, RawLink<uint>* links, LinksHeader<uint>* 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 uint</para>
48          /// <para></para>
49          /// </returns>
50          [MethodImpl(MethodImplOptions.AggressiveInlining)]
51          protected override ref uint GetLeftReference(uint node) => ref Links[node].LeftAsTarget;
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 uint</para>
65          /// <para></para>
66          /// </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>
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].SizeAsTarget;
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].SizeAsTarget = 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->RootAsTarget;
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].Target;
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                 => 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>
```

```
        /// </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;
        }
    }
}
```

1.99 ./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>
        /// <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>
```

```
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 }
```

1.100 ./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>
```

```
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
        ↪  DefaultLinksSizeStep) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt32UnitedMemoryLinks"/> 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 UInt32UnitedMemoryLinks(IResizableDirectMemory memory, long
        ↪  memoryReservationStep) : this(memory, memoryReservationStep,
        ↪  Default<LinksConstants<uint>>.Instance, IndexTreeType.Default) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt32UnitedMemoryLinks"/> 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>
        /// <param name="indexTreeType">
        /// <para>A index tree type.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32UnitedMemoryLinks(IResizableDirectMemory memory, long
        ↪  memoryReservationStep, LinksConstants<uint> constants, IndexTreeType indexTreeType)
        ↪  : base(memory, memoryReservationStep, constants)
        {
            if (indexTreeType == IndexTreeType.SizeBalancedTree)
            {
                _createSourceTreeMethods = () => new
                ↪  UInt32LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
                _createTargetTreeMethods = () => new
                ↪  UInt32LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
            }
            else
            {
                _createSourceTreeMethods = () => new
                ↪  UInt32LinksSourcesRecursionlessSizeBalancedTreeMethods(Constants, _links,
                ↪  _header);
                _createTargetTreeMethods = () => new
                ↪  UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods(Constants, _links,
                ↪  _header);
            }
            Init(memory, memoryReservationStep);
        }

        /// <summary>
        /// <para>
        /// Sets the pointers using the specified memory.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="memory">
```

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

        /// <summary>
        /// <para>
        /// Resets the pointers.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ResetPointers()
        {
            base.ResetPointers();
            _links = null;
            _header = null;
        }

        /// <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 index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="linkIndex">
        /// <para>The link index.</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 linkIndex) => ref
        ↪  _links[linkIndex];

        /// <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;
```

```
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>
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 GreaterThan(uint first, uint second) => first > second;
258
259         /// <summary>
260         /// <para>
261         /// Determines whether this instance greater or equal 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>
```

```
        /// <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>
        /// </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 Decrement(uint link) => --link;
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Memory.United.Specific
{
    /// <summary>
    /// <para>
    /// Represents the int 32 unused links list methods.
```

```
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="UnusedLinksListMethods{uint}"/>
    public unsafe class UInt32UnusedLinksListMethods : UnusedLinksListMethods<uint>
    {
        private readonly RawLink<uint>* _links;
        private readonly LinksHeader<uint>* _header;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UInt32UnusedLinksListMethods"/> 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 UInt32UnusedLinksListMethods(RawLink<uint>* links, LinksHeader<uint>* 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 uint</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLink<uint> GetLinkReference(uint link) => ref _links[link];

        /// <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;
    }
}
```

1.102 ./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() => 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)]
        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">
```

```
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>
376             /// </param>
377             /// <returns>
378             /// <para>The bool</para>
379             /// <para></para>
380             /// </returns>
381             [MethodImpl(MethodImplOptions.AggressiveInlining)]
382             protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
383             {
384                 ref var firstLink = ref Links[first];
385                 ref var secondLink = ref Links[second];
386                 return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
                     ↪  secondLink.Source, secondLink.Target);
387             }
388
389             /// <summary>
390             /// <para>
391             /// Gets the size value using the specified value.
392             /// </para>
393             /// <para></para>
394             /// </summary>
395             /// <param name="value">
396             /// <para>The value.</para>
397             /// <para></para>
398             /// </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);

        /// <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;

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

1.103 ./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">
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 UInt64LinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<ulong>
           ↪ constants, RawLink<ulong>* links, LinksHeader<ulong>* 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 ulong</para>
65         /// <para></para>
66         /// </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;
```

```
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(ulong first, ulong 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(ulong value) => true; // value >= 0 is
            ↪  always true for ulong
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(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 }
```

1.104 ./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
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the int 64 links size balanced tree methods base.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     /// <seealso cref="LinksSizeBalancedTreeMethodsBase{ulong}"/>
15     public unsafe abstract class UInt64LinksSizeBalancedTreeMethodsBase :
       ↪  LinksSizeBalancedTreeMethodsBase<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="UInt64LinksSizeBalancedTreeMethodsBase"/> 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 UInt64LinksSizeBalancedTreeMethodsBase(LinksConstants<ulong> constants,
            ↪ RawLink<ulong>* links, LinksHeader<ulong>* 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 ulong</para>
65          /// <para></para>
66          /// </returns>
67          [MethodImpl(MethodImplOptions.AggressiveInlining)]
68          protected override ulong GetZero() => 0UL;
69
70          /// <summary>
71          /// <para>
72          /// Determines whether this instance equal to zero.
73          /// </para>
74          /// <para></para>
75          /// </summary>
76          /// <param name="value">
77          /// <para>The value.</para>
78          /// <para></para>
79          /// </param>
80          /// <returns>
81          /// <para>The bool</para>
82          /// <para></para>
83          /// </returns>
84          [MethodImpl(MethodImplOptions.AggressiveInlining)]
85          protected override bool EqualToZero(ulong value) => value == 0UL;
86
87          /// <summary>
88          /// <para>
89          /// Determines whether this instance are equal.
90          /// </para>
91          /// <para></para>
92          /// </summary>
93          /// <param name="first">
94          /// <para>The first.</para>
95          /// <para></para>
96          /// </param>
97          /// <param name="second">
98          /// <para>The second.</para>
99          /// <para></para>
100         /// </param>
101         /// <returns>
102         /// <para>The bool</para>
103         /// <para></para>
104         /// </returns>
105         [MethodImpl(MethodImplOptions.AggressiveInlining)]
106         protected override bool AreEqual(ulong first, ulong second) => first == second;
107
```

```
        /// <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>
        /// 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 }
```

1.105 ./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.
195             /// </para>
196             /// <para></para>
197             /// </summary>
198             /// <param name="node">
199             /// <para>The node.</para>
200             /// <para></para>
201             /// </param>
202             /// <param name="value">
203             /// <para>The value.</para>
204             /// <para></para>
205             /// </param>
206             [MethodImpl(MethodImplOptions.AggressiveInlining)]
207             protected override void SetLeftIsChild(ulong node, bool value) =>
                ↪  SetLeftIsChildValue(ref Links[node].SizeAsSource, value);
208
209             /// <summary>
210             /// <para>
211             /// Determines whether this instance get right is child.
212             /// </para>
213             /// <para></para>
214             /// </summary>
215             /// <param name="node">
216             /// <para>The node.</para>
217             /// <para></para>
```

```
218         /// </param>
219         /// <returns>
220         /// <para>The bool</para>
221         /// <para></para>
222         /// </returns>
223         [MethodImpl(MethodImplOptions.AggressiveInlining)]
224         protected override bool GetRightIsChild(ulong node) =>
            ↪ GetRightIsChildValue(Links[node].SizeAsSource);
225
226         //[MethodImpl(MethodImplOptions.AggressiveInlining)]
227         //protected override bool GetRightIsChild(ulong node) => IsChild(node, GetRight(node));
228
229         /// <summary>
230         /// <para>
231         /// Sets the right is child 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         /// <param name="value">
240         /// <para>The value.</para>
241         /// <para></para>
242         /// </param>
243         [MethodImpl(MethodImplOptions.AggressiveInlining)]
244         protected override void SetRightIsChild(ulong node, bool value) =>
            ↪ SetRightIsChildValue(ref Links[node].SizeAsSource, value);
245
246         /// <summary>
247         /// <para>
248         /// Gets the balance using the specified node.
249         /// </para>
250         /// <para></para>
251         /// </summary>
252         /// <param name="node">
253         /// <para>The node.</para>
254         /// <para></para>
255         /// </param>
256         /// <returns>
257         /// <para>The sbyte</para>
258         /// <para></para>
259         /// </returns>
260         [MethodImpl(MethodImplOptions.AggressiveInlining)]
261         protected override sbyte GetBalance(ulong node) =>
            ↪ GetBalanceValue(Links[node].SizeAsSource);
262
263         /// <summary>
264         /// <para>
265         /// Sets the balance using the specified node.
266         /// </para>
267         /// <para></para>
268         /// </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 = OUL;
385             link.RightAsSource = OUL;
386             link.SizeAsSource = OUL;
387         }
388     }
389 }
```

1.106 ./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>
```

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

        /// <summary>
        /// <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>
193         /// <para></para>
194         /// </param>
195         /// <returns>
196         /// <para>The ulong</para>
197         /// <para></para>
198         /// </returns>
```

```
199             [MethodImpl(MethodImplOptions.AggressiveInlining)]
200             protected override ulong GetBasePartValue(ulong 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(ulong firstSource, ulong firstTarget,
                ↪  ulong secondSource, ulong 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(ulong firstSource, ulong firstTarget,
                ↪  ulong secondSource, ulong 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(ulong node)
274         {
275             ref var link = ref Links[node];
276             link.LeftAsSource = OUL;
277             link.RightAsSource = OUL;
278             link.SizeAsSource = OUL;
279         }
280     }
281 }
```

1.107 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesSizeBalancedTreeMethods.c

```
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 size balanced tree methods.
10     /// </para>
11     /// <para></para>
12     /// </summary>
13     /// <seealso cref="UInt64LinksSizeBalancedTreeMethodsBase"/>
14     public unsafe class UInt64LinksSourcesSizeBalancedTreeMethods :
       ↪  UInt64LinksSizeBalancedTreeMethodsBase
15     {
16         /// <summary>
17         /// <para>
18         /// Initializes a new <see cref="UInt64LinksSourcesSizeBalancedTreeMethods"/> 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 UInt64LinksSourcesSizeBalancedTreeMethods(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>
```

```
        /// <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>
        [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 = 0UL;
            link.RightAsSource = 0UL;
            link.SizeAsSource = 0UL;
        }
    }
}
```

1.108 ./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>
        /// <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) => GetSizeValue(Links[node].SizeAsTarget);

```

```
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].SizeAsTarget, 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].SizeAsTarget);
188
189         /// <summary>
190         /// <para>
191         /// Sets the left is child using the specified node.
192         /// </para>
193         /// <para></para>
194         /// </summary>
195         /// <param name="node">
196         /// <para>The node.</para>
197         /// <para></para>
198         /// </param>
199         /// <param name="value">
200         /// <para>The value.</para>
201         /// <para></para>
202         /// </param>
203         [MethodImpl(MethodImplOptions.AggressiveInlining)]
204         protected override void SetLeftIsChild(ulong node, bool value) =>
            ↪ SetLeftIsChildValue(ref Links[node].SizeAsTarget, value);
205
206         /// <summary>
207         /// <para>
208         /// Determines whether this instance get right is child.
209         /// </para>
210         /// <para></para>
211         /// </summary>
212         /// <param name="node">
213         /// <para>The node.</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 GetRightIsChild(ulong node) =>
            ↪ GetRightIsChildValue(Links[node].SizeAsTarget);
222
223         /// <summary>
224         /// <para>
225         /// Sets the right is child using the specified node.
226         /// </para>
227         /// <para></para>
228         /// </summary>
```

```
229             /// <param name="node">
230             /// <para>The node.</para>
231             /// <para></para>
232             /// </param>
233             /// <param name="value">
234             /// <para>The value.</para>
235             /// <para></para>
236             /// </param>
237             [MethodImpl(MethodImplOptions.AggressiveInlining)]
238             protected override void SetRightIsChild(ulong node, bool value) =>
                ↪ SetRightIsChildValue(ref Links[node].SizeAsTarget, value);
239
240             /// <summary>
241             /// <para>
242             /// Gets the balance using the specified node.
243             /// </para>
244             /// <para></para>
245             /// </summary>
246             /// <param name="node">
247             /// <para>The node.</para>
248             /// <para></para>
249             /// </param>
250             /// <returns>
251             /// <para>The sbyte</para>
252             /// <para></para>
253             /// </returns>
254             [MethodImpl(MethodImplOptions.AggressiveInlining)]
255             protected override sbyte GetBalance(ulong node) =>
                ↪ GetBalanceValue(Links[node].SizeAsTarget);
256
257             /// <summary>
258             /// <para>
259             /// Sets the balance using the specified node.
260             /// </para>
261             /// <para></para>
262             /// </summary>
263             /// <param name="node">
264             /// <para>The node.</para>
265             /// <para></para>
266             /// </param>
267             /// <param name="value">
268             /// <para>The value.</para>
269             /// <para></para>
270             /// </param>
271             [MethodImpl(MethodImplOptions.AggressiveInlining)]
272             protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
                ↪ Links[node].SizeAsTarget, value);
273
274             /// <summary>
275             /// <para>
276             /// Gets the tree root.
277             /// </para>
278             /// <para></para>
279             /// </summary>
280             /// <returns>
281             /// <para>The ulong</para>
282             /// <para></para>
283             /// </returns>
284             [MethodImpl(MethodImplOptions.AggressiveInlining)]
285             protected override ulong GetTreeRoot() => Header->RootAsTarget;
286
287             /// <summary>
288             /// <para>
289             /// Gets the base part value using the specified link.
290             /// </para>
291             /// <para></para>
292             /// </summary>
293             /// <param name="link">
294             /// <para>The link.</para>
295             /// <para></para>
296             /// </param>
297             /// <returns>
298             /// <para>The ulong</para>
299             /// <para></para>
300             /// </returns>
301             [MethodImpl(MethodImplOptions.AggressiveInlining)]
302             protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
303
```

```
304         /// <summary>
305         /// <para>
306         /// Determines whether this instance first is to the left of second.
307         /// </para>
308         /// <para></para>
309         /// </summary>
310         /// <param name="firstSource">
311         /// <para>The first source.</para>
312         /// <para></para>
313         /// </param>
314         /// <param name="firstTarget">
315         /// <para>The first target.</para>
316         /// <para></para>
317         /// </param>
318         /// <param name="secondSource">
319         /// <para>The second source.</para>
320         /// <para></para>
321         /// </param>
322         /// <param name="secondTarget">
323         /// <para>The second target.</para>
324         /// <para></para>
325         /// </param>
326         /// <returns>
327         /// <para>The bool</para>
328         /// <para></para>
329         /// </returns>
330         [MethodImpl(MethodImplOptions.AggressiveInlining)]
331         protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
          ↪  ulong secondSource, ulong secondTarget)
332             => firstTarget < secondTarget || (firstTarget == secondTarget && 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(ulong firstSource, ulong firstTarget,
          ↪  ulong secondSource, ulong secondTarget)
362             => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
              ↪  secondSource);
363 
364         /// <summary>
365         /// <para>
366         /// Clears the node using the specified node.
367         /// </para>
368         /// <para></para>
369         /// </summary>
370         /// <param name="node">
371         /// <para>The node.</para>
372         /// <para></para>
373         /// </param>
374         [MethodImpl(MethodImplOptions.AggressiveInlining)]
375         protected override void ClearNode(ulong node)
376         {
```

```
            ref var link = ref Links[node];
            link.LeftAsTarget = OUL;
            link.RightAsTarget = OUL;
            link.SizeAsTarget = OUL;
        }
    }
}
```

1.109 ./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>
```

```
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>
134         /// </param>
135         [MethodImpl(MethodImplOptions.AggressiveInlining)]
136         protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
            ↪  right;
137
138         /// <summary>
139         /// <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 = 0UL;
            link.RightAsTarget = 0UL;
            link.SizeAsTarget = 0UL;
        }
    }
}
```

1.110 ./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>
```

```
8       /// <para>
9       /// Represents the int 64 links targets size balanced tree methods.
10      /// </para>
11      /// <para></para>
12      /// </summary>
13      /// <seealso cref="UInt64LinksSizeBalancedTreeMethodsBase"/>
14      public unsafe class UInt64LinksTargetsSizeBalancedTreeMethods :
        ↪  UInt64LinksSizeBalancedTreeMethodsBase
15      {
16          /// <summary>
17          /// <para>
18          /// Initializes a new <see cref="UInt64LinksTargetsSizeBalancedTreeMethods"/> 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 UInt64LinksTargetsSizeBalancedTreeMethods(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].LeftAsTarget;
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].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>
```

```
        /// <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>
```

```
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].SizeAsTarget =
            ↪ 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->RootAsTarget;
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 ulong</para>
197         /// <para></para>
198         /// </returns>
199         [MethodImpl(MethodImplOptions.AggressiveInlining)]
200         protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
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(ulong firstSource, ulong firstTarget,
            ↪ ulong secondSource, ulong secondTarget)
                => 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 }
```

1.111 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnitedMemoryLinks.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="ulong"
       ↪  />.</para>
13     /// <para>Представляет низкоуровневую реализация прямого доступа к памяти с переменным
       ↪  размером, для организации хранения связей с адресами представленными в виде <see
       ↪  cref="ulong"/>.</para>
14     /// </summary>
15     public unsafe class UInt64UnitedMemoryLinks : UnitedMemoryLinksBase<ulong>
16     {
17         private readonly Func<ILinksTreeMethods<ulong>> _createSourceTreeMethods;
18         private readonly Func<ILinksTreeMethods<ulong>> _createTargetTreeMethods;
19         private LinksHeader<ulong>* _header;
20         private RawLink<ulong>* _links;
21 
```

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

        /// <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>
        /// </param>
        /// <param name="constants">
        /// <para>A constants.</para>
        /// <para></para>
        /// </param>
        /// <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>
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 LessThan(ulong first, ulong second) => first < second;
221 
222         /// <summary>
223         /// <para>
224         /// Determines whether this instance less or equal than.
225         /// </para>
226         /// <para></para>
227         /// </summary>
228         /// <param name="first">
229         /// <para>The first.</para>
230         /// <para></para>
231         /// </param>
232         /// <param name="second">
233         /// <para>The second.</para>
234         /// <para></para>
235         /// </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;
    }
}
```

1.112 ./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;
    }
}
```

1.113 ./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(links.GetIndex(valueLink));
        }

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

1.114 ./csharp/Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs

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

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

namespace Platform.Data.Doublets.PropertyOperators
{
    /// <summary>
    /// <para>
    /// Represents the property operator.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
    /// <seealso cref="IProperty{TLinkAddress, TLinkAddress}"/>
    public class PropertyOperator<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
    ↪  IProperty<TLinkAddress, TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;
        private readonly TLinkAddress _propertyMarker;
        private readonly TLinkAddress _propertyValueMarker;

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

```
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="propertyMarker">
        /// <para>A property marker.</para>
        /// <para></para>
        /// </param>
        /// <param name="propertyValueMarker">
        /// <para>A property value marker.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public PropertyOperator(ILinks<TLinkAddress> links, TLinkAddress propertyMarker,
        ↪  TLinkAddress propertyValueMarker) : base(links)
        {
            _propertyMarker = propertyMarker;
            _propertyValueMarker = propertyValueMarker;
        }

        /// <summary>
        /// <para>
        /// Gets the 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 Get(TLinkAddress link)
        {
            var property = _links.SearchOrDefault(link, _propertyMarker);
            return GetValue(GetContainer(property));
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private TLinkAddress GetContainer(TLinkAddress property)
        {
            var valueContainer = default(TLinkAddress);
            if (_equalityComparer.Equals(property, default))
            {
                return valueContainer;
            }
            var links = _links;
            var constants = links.Constants;
            var countinueConstant = constants.Continue;
            var breakConstant = constants.Break;
            var anyConstant = constants.Any;
            var query = new Link<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);
            }
        }
    }
}
```

1.115 ./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)]
```

```
50          public Stack(ILinks<TLinkAddress> links, TLinkAddress stack) : base(links) => _stack =
            ↪  stack;
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          private TLinkAddress GetStackMarker() => _links.GetSource(_stack);
53          [MethodImpl(MethodImplOptions.AggressiveInlining)]
54          private TLinkAddress GetTop() => _links.GetTarget(_stack);
55
56          /// <summary>
57          /// <para>
58          /// Peeks this instance.
59          /// </para>
60          /// <para></para>
61          /// </summary>
62          /// <returns>
63          /// <para>The link</para>
64          /// <para></para>
65          /// </returns>
66          [MethodImpl(MethodImplOptions.AggressiveInlining)]
67          public TLinkAddress Peek() => _links.GetTarget(GetTop());
68
69          /// <summary>
70          /// <para>
71          /// Pops this instance.
72          /// </para>
73          /// <para></para>
74          /// </summary>
75          /// <returns>
76          /// <para>The element.</para>
77          /// <para></para>
78          /// </returns>
79          [MethodImpl(MethodImplOptions.AggressiveInlining)]
80          public TLinkAddress Pop()
81          {
82              var element = Peek();
83              if (!_equalityComparer.Equals(element, _stack))
84              {
85                  var top = GetTop();
86                  var previousTop = _links.GetSource(top);
87                  _links.Update(_stack, GetStackMarker(), previousTop);
88                  _links.Delete(top);
89              }
90              return element;
91          }
92
93          /// <summary>
94          /// <para>
95          /// Pushes the element.
96          /// </para>
97          /// <para></para>
98          /// </summary>
99          /// <param name="element">
100         /// <para>The element.</para>
101         /// <para></para>
102         /// </param>
103         [MethodImpl(MethodImplOptions.AggressiveInlining)]
104         public void Push(TLinkAddress element) => _links.Update(_stack, GetStackMarker(),
            ↪  _links.GetOrCreate(GetTop(), element));
105     }
106 }
```

1.116 ./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.
```

```
        /// </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="stackMarker">
        /// <para>The stack marker.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The stack.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLinkAddress CreateStack<TLinkAddress>(this ILinks<TLinkAddress> links,
        ↪  TLinkAddress stackMarker)
        {
            var stackPoint = links.CreatePoint();
            var stack = links.Update(stackPoint, stackMarker, stackPoint);
            return stack;
        }
    }
}
```

1.117 ./csharp/Platform.Data.Doublets/SynchronizedLinks.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Data.Doublets;
using Platform.Delegates;
using Platform.Threading.Synchronization;

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

namespace Platform.Data.Doublets
{
    /// <remarks>
    /// TODO: Autogeneration of synchronized wrapper (decorator).
    /// TODO: Try to unfold code of each method using IL generation for performance improvements.
    /// TODO: Or even to unfold multiple layers of implementations.
    /// </remarks>
    public class SynchronizedLinks<TLinkAddress> : ISynchronizedLinks<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Gets the constants value.
        /// </para>
        /// <para></para>
        /// </summary>
        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>
```

```
49          public ILinks<TLinkAddress> Sync
50          {
51              [MethodImpl(MethodImplOptions.AggressiveInlining)]
52              get;
53          }
54
55          /// <summary>
56          /// <para>
57          /// Gets the unsync value.
58          /// </para>
59          /// <para></para>
60          /// </summary>
61          public ILinks<TLinkAddress> Unsync
62          {
63              [MethodImpl(MethodImplOptions.AggressiveInlining)]
64              get;
65          }
66
67          /// <summary>
68          /// <para>
69          /// Initializes a new <see cref="SynchronizedLinks"/> instance.
70          /// </para>
71          /// <para></para>
72          /// </summary>
73          /// <param name="links">
74          /// <para>A links.</para>
75          /// <para></para>
76          /// </param>
77          [MethodImpl(MethodImplOptions.AggressiveInlining)]
78          public SynchronizedLinks(ILinks<TLinkAddress> links) : this(new
            ↪  ReaderWriterLockSynchronization(), links) { }
79
80          /// <summary>
81          /// <para>
82          /// Initializes a new <see cref="SynchronizedLinks"/> instance.
83          /// </para>
84          /// <para></para>
85          /// </summary>
86          /// <param name="synchronization">
87          /// <para>A synchronization.</para>
88          /// <para></para>
89          /// </param>
90          /// <param name="links">
91          /// <para>A links.</para>
92          /// <para></para>
93          /// </param>
94          [MethodImpl(MethodImplOptions.AggressiveInlining)]
95          public SynchronizedLinks(ISynchronization synchronization, ILinks<TLinkAddress> links)
96          {
97              SyncRoot = synchronization;
98              Sync = this;
99              Unsync = links;
100             Constants = links.Constants;
101         }
102
103         /// <summary>
104         /// <para>
105         /// Counts the restriction.
106         /// </para>
107         /// <para></para>
108         /// </summary>
109         /// <param name="restriction">
110         /// <para>The restriction.</para>
111         /// <para></para>
112         /// </param>
113         /// <returns>
114         /// <para>The link address</para>
115         /// <para></para>
116         /// </returns>
117         [MethodImpl(MethodImplOptions.AggressiveInlining)]
118         public TLinkAddress Count(IList<TLinkAddress>? restriction) =>
            ↪  SyncRoot.DoRead(restriction, Unsync.Count);
119
120         /// <summary>
121         /// <para>
122         /// Eaches the handler.
123         /// </para>
124         /// <para></para>
```

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

1.118 ./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         {
```

```
            var sb = new StringBuilder();
            var visited = new HashSet<ulong>();
            links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
            ↪  renderDebug);
            return sb.ToString();
        }

        /// <summary>
        /// <para>
        /// Appends the structure using the specified links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>The links.</para>
        /// <para></para>
        /// </param>
        /// <param name="sb">
        /// <para>The sb.</para>
        /// <para></para>
        /// </param>
        /// <param name="visited">
        /// <para>The visited.</para>
        /// <para></para>
        /// </param>
        /// <param name="linkIndex">
        /// <para>The link index.</para>
        /// <para></para>
        /// </param>
        /// <param name="isElement">
        /// <para>The is element.</para>
        /// <para></para>
        /// </param>
        /// <param name="appendElement">
        /// <para>The append element.</para>
        /// <para></para>
        /// </param>
        /// <param name="renderIndex">
        /// <para>The render index.</para>
        /// <para></para>
        /// </param>
        /// <param name="renderDebug">
        /// <para>The render debug.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="ArgumentNullException">
        /// <para></para>
        /// <para></para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void AppendStructure(this ILinks<ulong> links, StringBuilder sb,
        ↪  HashSet<ulong> visited, ulong linkIndex, Func<Link<ulong>, bool> isElement,
        ↪  Action<StringBuilder, Link<ulong>> appendElement, bool renderIndex = false, bool
        ↪  renderDebug = false)
        {
            if (sb == null)
            {
                throw new ArgumentNullException(nameof(sb));
            }
            if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
            ↪  Constants.Itself)
            {
                return;
            }
            if (links.Exists(linkIndex))
            {
                if (visited.Add(linkIndex))
                {
                    sb.Append('(');
                    var link = new Link<ulong>(links.GetLink(linkIndex));
                    if (renderIndex)
                    {
                        sb.Append(link.Index);
                        sb.Append(':');
                    }
                    if (link.Source == link.Index)
                    {
                        sb.Append(link.Index);
```

```
                    }
                    else
                    {
                        var source = new Link<ulong>(links.GetLink(link.Source));
                        if (isElement(source))
                        {
                            appendElement(sb, source);
                        }
                        else
                        {
                            links.AppendStructure(sb, visited, source.Index, isElement,
                                ↪  appendElement, renderIndex);
                        }
                    }
                    sb.Append(' ');
                    if (link.Target == link.Index)
                    {
                        sb.Append(link.Index);
                    }
                    else
                    {
                        var target = new Link<ulong>(links.GetLink(link.Target));
                        if (isElement(target))
                        {
                            appendElement(sb, target);
                        }
                        else
                        {
                            links.AppendStructure(sb, visited, target.Index, isElement,
                                ↪  appendElement, renderIndex);
                        }
                    }
                    sb.Append(')');
                }
                else
                {
                    if (renderDebug)
                    {
                        sb.Append('*');
                    }
                    sb.Append(linkIndex);
                }
            }
            else
            {
                if (renderDebug)
                {
                    sb.Append('~');
                }
                sb.Append(linkIndex);
            }
        }
    }
}
```

1.119 ./csharp/Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs

```
using System;
using System.Linq;
using System.Collections.Generic;
using System.IO;
using System.Runtime.CompilerServices;
using System.Threading;
using System.Threading.Tasks;
using Platform.Disposables;
using Platform.Timestamps;
using Platform.Unsafe;
using Platform.IO;
using Platform.Data.Doublets.Decorators;
using Platform.Delegates;
using Platform.Exceptions;
using TLinkAddress = System.UInt64;

#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>
```

```
25      /// <para></para>
26      /// </summary>
27      /// <seealso cref="LinksDisposableDecoratorBase{TLinkAddress}"/>
28      public class UInt64LinksTransactionsLayer : LinksDisposableDecoratorBase<TLinkAddress>
        ↪ //-V3073
29      {
30          /// <remarks>
31          /// Альтернативные варианты хранения трансформации (элемента транзакции):
32          ///
33          /// private enum TransitionType
34          /// {
35          ///     Creation,
36          ///     UpdateOf,
37          ///     UpdateTo,
38          ///     Deletion
39          /// }
40          ///
41          /// private struct Transition
42          /// {
43          ///     public TLinkAddress TransactionId;
44          ///     public UniqueTimestamp Timestamp;
45          ///     public TransactionItemType Type;
46          ///     public Link Source;
47          ///     public Link Linker;
48          ///     public Link Target;
49          /// }
50          ///
51          /// Или
52          ///
53          /// public struct TransitionHeader
54          /// {
55          ///     public TLinkAddress TransactionIdCombined;
56          ///     public TLinkAddress TimestampCombined;
57          ///
58          ///     public TLinkAddress TransactionId
59          ///     {
60          ///         get
61          ///         {
62          ///             return (TLinkAddress) mask &amp; TransactionIdCombined;
63          ///         }
64          ///     }
65          ///
66          ///     public UniqueTimestamp Timestamp
67          ///     {
68          ///         get
69          ///         {
70          ///             return (UniqueTimestamp)mask &amp; TransactionIdCombined;
71          ///         }
72          ///     }
73          ///
74          ///     public TransactionItemType Type
75          ///     {
76          ///         get
77          ///         {
78          ///             // Использовать по одному биту из TransactionId и Timestamp,
79          ///             // для значения в 2 бита, которое представляет тип операции
80          ///             throw new NotImplementedException();
81          ///         }
82          ///     }
83          /// }
84          ///
85          /// private struct Transition
86          /// {
87          ///     public TransitionHeader Header;
88          ///     public Link Source;
89          ///     public Link Linker;
90          ///     public Link Target;
91          /// }
92          ///
93          /// </remarks>
94          public struct Transition : IEquatable<Transition>
95          {
96              /// <summary>
97              /// <para>
98              /// The size.
99              /// </para>
100             /// <para></para>
101             /// </summary>
```

```
102             public static readonly long Size = Structure<Transition>.Size;
103
104             /// <summary>
105             /// <para>
106             /// The transaction id.
107             /// </para>
108             /// <para></para>
109             /// </summary>
110             public readonly TLinkAddress TransactionId;
111             /// <summary>
112             /// <para>
113             /// The before.
114             /// </para>
115             /// <para></para>
116             /// </summary>
117             public readonly Link<TLinkAddress> Before;
118             /// <summary>
119             /// <para>
120             /// The after.
121             /// </para>
122             /// <para></para>
123             /// </summary>
124             public readonly Link<TLinkAddress> After;
125             /// <summary>
126             /// <para>
127             /// The timestamp.
128             /// </para>
129             /// <para></para>
130             /// </summary>
131             public readonly Timestamp Timestamp;
132
133             /// <summary>
134             /// <para>
135             /// Initializes a new <see cref="Transition"/> instance.
136             /// </para>
137             /// <para></para>
138             /// </summary>
139             /// <param name="uniqueTimestampFactory">
140             /// <para>A unique timestamp factory.</para>
141             /// <para></para>
142             /// </param>
143             /// <param name="transactionId">
144             /// <para>A transaction id.</para>
145             /// <para></para>
146             /// </param>
147             /// <param name="before">
148             /// <para>A before.</para>
149             /// <para></para>
150             /// </param>
151             /// <param name="after">
152             /// <para>A after.</para>
153             /// <para></para>
154             /// </param>
155             [MethodImpl(MethodImplOptions.AggressiveInlining)]
156             public Transition(UniqueTimestampFactory uniqueTimestampFactory, TLinkAddress
                ↪  transactionId, Link<TLinkAddress> before, Link<TLinkAddress> after)
157             {
158                 TransactionId = transactionId;
159                 Before = before;
160                 After = after;
161                 Timestamp = uniqueTimestampFactory.Create();
162             }
163
164             public Transition(UniqueTimestampFactory uniqueTimestampFactory, TLinkAddress
                ↪  transactionId, IList<TLinkAddress> before, IList<TLinkAddress> after) :
                ↪  this(uniqueTimestampFactory, transactionId, new Link<TLinkAddress>(before), new
                ↪  Link<TLinkAddress>(after)) { }
165
166             /// <summary>
167             /// <para>
168             /// Initializes a new <see cref="Transition"/> instance.
169             /// </para>
170             /// <para></para>
171             /// </summary>
172             /// <param name="uniqueTimestampFactory">
173             /// <para>A unique timestamp factory.</para>
174             /// <para></para>
175             /// </param>
```

```
176                 /// <param name="transactionId">
177                 /// <para>A transaction id.</para>
178                 /// <para></para>
179                 /// </param>
180                 /// <param name="before">
181                 /// <para>A before.</para>
182                 /// <para></para>
183                 /// </param>
184                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
185                 public Transition(UniqueTimestampFactory uniqueTimestampFactory, TLinkAddress
                    ↪ transactionId, Link<TLinkAddress> before) : this(uniqueTimestampFactory,
                    ↪ transactionId, before, default) { }
186
187                 /// <summary>
188                 /// <para>
189                 /// Initializes a new <see cref="Transition"/> instance.
190                 /// </para>
191                 /// <para></para>
192                 /// </summary>
193                 /// <param name="uniqueTimestampFactory">
194                 /// <para>A unique timestamp factory.</para>
195                 /// <para></para>
196                 /// </param>
197                 /// <param name="transactionId">
198                 /// <para>A transaction id.</para>
199                 /// <para></para>
200                 /// </param>
201                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
202                 public Transition(UniqueTimestampFactory uniqueTimestampFactory, TLinkAddress
                    ↪ transactionId) : this(uniqueTimestampFactory, transactionId, default, default) {
                    ↪ }
203
204                 /// <summary>
205                 /// <para>
206                 /// Returns the string.
207                 /// </para>
208                 /// <para></para>
209                 /// </summary>
210                 /// <returns>
211                 /// <para>The string</para>
212                 /// <para></para>
213                 /// </returns>
214                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
215                 public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
                    ↪ {After}";
216
217                 /// <summary>
218                 /// <para>
219                 /// Determines whether this instance equals.
220                 /// </para>
221                 /// <para></para>
222                 /// </summary>
223                 /// <param name="obj">
224                 /// <para>The obj.</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 obj) => obj is Transition transition ?
                    ↪ Equals(transition) : false;
233
234                 /// <summary>
235                 /// <para>
236                 /// Gets the hash code.
237                 /// </para>
238                 /// <para></para>
239                 /// </summary>
240                 /// <returns>
241                 /// <para>The int</para>
242                 /// <para></para>
243                 /// </returns>
244                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
245                 public override int GetHashCode() => (TransactionId, Before, After,
                    ↪ Timestamp).GetHashCode();
```

```
246
247             /// <summary>
248             /// <para>
249             /// Determines whether this instance equals.
250             /// </para>
251             /// <para></para>
252             /// </summary>
253             /// <param name="other">
254             /// <para>The other.</para>
255             /// <para></para>
256             /// </param>
257             /// <returns>
258             /// <para>The bool</para>
259             /// <para></para>
260             /// </returns>
261             [MethodImpl(MethodImplOptions.AggressiveInlining)]
262             public bool Equals(Transition other) => TransactionId == other.TransactionId &&
                ↪  Before == other.Before && After == other.After && Timestamp == other.Timestamp;
263
264             [MethodImpl(MethodImplOptions.AggressiveInlining)]
265             public static bool operator ==(Transition left, Transition right) =>
                ↪  left.Equals(right);
266
267             [MethodImpl(MethodImplOptions.AggressiveInlining)]
268             public static bool operator !=(Transition left, Transition right) => !(left ==
                ↪  right);
269         }
270
271         /// <remarks>
272         /// Другие варианты реализации транзакций (атомарности):
273         ///     1. Разделение хранения значения связи ((Source Target) или (Source Linker
            ↪  Target)) и индексов.
274         ///     2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
            ↪  потребуется решить вопрос
275         ///        со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
            ↪  пересечениями идентификаторов.
276         ///
277         /// Где хранить промежуточный список транзакций?
278         ///
279         /// В оперативной памяти:
280         ///  Минусы:
281         ///     1. Может усложнить систему, если она будет функционировать самостоятельно,
282         ///     так как нужно отдельно выделять память под список трансформаций.
283         ///     2. Выделенной оперативной памяти может не хватить, в том случае,
284         ///     если транзакция использует слишком много трансформаций.
285         ///         -> Можно использовать жёсткий диск для слишком длинных транзакций.
286         ///         -> Максимальный размер списка трансформаций можно ограничить / задать
            ↪  константой.
287         ///     3. При подтверждении транзакции (Commit) все трансформации записываются разом
            ↪  создавая задержку.
288         ///
289         /// На жёстком диске:
290         ///  Минусы:
291         ///     1. Длительный отклик, на запись каждой трансформации.
292         ///     2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
293         ///         -> Это может решаться упаковкой/исключением дублирующих операций.
294         ///         -> Также это может решаться тем, что короткие транзакции вообще
295         ///            не будут записываться в случае отката.
296         ///     3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
            ↪  операции (трансформации)
297         ///        будут записаны в лог.
298         ///
299         /// </remarks>
300         public class Transaction : DisposableBase
301         {
302             private readonly Queue<Transition> _transitions;
303             private readonly UInt64LinksTransactionsLayer _layer;
304             /// <summary>
305             /// <para>
306             /// Gets or sets the is committed value.
307             /// </para>
308             /// <para></para>
309             /// </summary>
310             public bool IsCommitted { get; private set; }
311             /// <summary>
312             /// <para>
313             /// Gets or sets the is reverted value.
314             /// </para>
```

```
        /// <para></para>
        /// </summary>
        public bool IsReverted { get; private set; }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Transaction"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="layer">
        /// <para>A layer.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="NotSupportedException">
        /// <para>Nested transactions not supported.</para>
        /// <para></para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Transaction(UInt64LinksTransactionsLayer layer)
        {
            _layer = layer;
            if (_layer._currentTransactionId != 0)
            {
                throw new NotSupportedException("Nested transactions not supported.");
            }
            IsCommitted = false;
            IsReverted = false;
            _transitions = new Queue<Transition>();
            SetCurrentTransaction(layer, this);
        }

        /// <summary>
        /// <para>
        /// Commits this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Commit()
        {
            EnsureTransactionAllowsWriteOperations(this);
            while (_transitions.Count > 0)
            {
                var transition = _transitions.Dequeue();
                _layer._transitions.Enqueue(transition);
            }
            _layer._lastCommitedTransactionId = _layer._currentTransactionId;
            IsCommitted = true;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void Revert()
        {
            EnsureTransactionAllowsWriteOperations(this);
            var transitionsToRevert = new Transition[_transitions.Count];
            _transitions.CopyTo(transitionsToRevert, 0);
            for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
            {
                _layer.RevertTransition(transitionsToRevert[i]);
            }
            IsReverted = true;
        }

        /// <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)]
```

```
393             public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
                ↪  Transaction transaction)
394             {
395                 layer._currentTransactionId = layer._lastCommitedTransactionId + 1;
396                 layer._currentTransactionTransitions = transaction._transitions;
397                 layer._currentTransaction = transaction;
398             }
399 
400             /// <summary>
401             /// <para>
402             /// Ensures the transaction allows write operations using the specified transaction.
403             /// </para>
404             /// <para></para>
405             /// </summary>
406             /// <param name="transaction">
407             /// <para>The transaction.</para>
408             /// <para></para>
409             /// </param>
410             /// <exception cref="InvalidOperationException">
411             /// <para>Transation is commited.</para>
412             /// <para></para>
413             /// </exception>
414             /// <exception cref="InvalidOperationException">
415             /// <para>Transation is reverted.</para>
416             /// <para></para>
417             /// </exception>
418             [MethodImpl(MethodImplOptions.AggressiveInlining)]
419             public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
420             {
421                 if (transaction.IsReverted)
422                 {
423                     throw new InvalidOperationException("Transation is reverted.");
424                 }
425                 if (transaction.IsCommitted)
426                 {
427                     throw new InvalidOperationException("Transation is commited.");
428                 }
429             }
430 
431             /// <summary>
432             /// <para>
433             /// Disposes the manual.
434             /// </para>
435             /// <para></para>
436             /// </summary>
437             /// <param name="manual">
438             /// <para>The manual.</para>
439             /// <para></para>
440             /// </param>
441             /// <param name="wasDisposed">
442             /// <para>The was disposed.</para>
443             /// <para></para>
444             /// </param>
445             [MethodImpl(MethodImplOptions.AggressiveInlining)]
446             protected override void Dispose(bool manual, bool wasDisposed)
447             {
448                 if (!wasDisposed && _layer != null && !_layer.Disposable.IsDisposed)
449                 {
450                     if (!IsCommitted && !IsReverted)
451                     {
452                         Revert();
453                     }
454                     _layer.ResetCurrentTransation();
455                 }
456             }
457         }
458 
459         /// <summary>
460         /// <para>
461         /// The from seconds.
462         /// </para>
463         /// <para></para>
464         /// </summary>
465         public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
466         private readonly string _logAddress;
467         private readonly FileStream _log;
468         private readonly Queue<Transition> _transitions;
469         private readonly UniqueTimestampFactory _uniqueTimestampFactory;
```

```
470         private Task _transitionsPusher;
471         private Transition _lastCommitedTransition;
472         private TLinkAddress _currentTransactionId;
473         private Queue<Transition> _currentTransactionTransitions;
474         private Transaction _currentTransaction;
475         private TLinkAddress _lastCommitedTransactionId;
476 
477         /// <summary>
478         /// <para>
479         /// Initializes a new <see cref="UInt64LinksTransactionsLayer"/> instance.
480         /// </para>
481         /// <para></para>
482         /// </summary>
483         /// <param name="links">
484         /// <para>A links.</para>
485         /// <para></para>
486         /// </param>
487         /// <param name="logAddress">
488         /// <para>A log address.</para>
489         /// <para></para>
490         /// </param>
491         /// <exception cref="ArgumentNullException">
492         /// <para></para>
493         /// <para></para>
494         /// </exception>
495         /// <exception cref="NotSupportedException">
496         /// <para>Database is damaged, autorecovery is not supported yet.</para>
497         /// <para></para>
498         /// </exception>
499         [MethodImpl(MethodImplOptions.AggressiveInlining)]
500         public UInt64LinksTransactionsLayer(ILinks<TLinkAddress> links, string logAddress)
501             : base(links)
502         {
503             if (string.IsNullOrWhiteSpace(logAddress))
504             {
505                 throw new ArgumentNullException(nameof(logAddress));
506             }
507             // В первой строке файла хранится последняя закоммиченную транзакцию.
508             // При запуске это используется для проверки удачного закрытия файла лога.
509             // In the first line of the file the last committed transaction is stored.
510             // On startup, this is used to check that the log file is successfully closed.
511             var lastCommitedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
512             var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
513             if (!lastCommitedTransition.Equals(lastWrittenTransition))
514             {
515                 Dispose();
516                 throw new NotSupportedException("Database is damaged, autorecovery is not
                    ↪  supported yet.");
517             }
518             if (lastCommitedTransition == default)
519             {
520                 FileHelpers.WriteFirst(logAddress, lastCommitedTransition);
521             }
522             _lastCommitedTransition = lastCommitedTransition;
523             // TODO: Think about a better way to calculate or store this value
524             var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
525             _lastCommitedTransactionId = allTransitions.Length > 0 ? allTransitions.Max(x =>
                ↪  x.TransactionId) : 0;
526             _uniqueTimestampFactory = new UniqueTimestampFactory();
527             _logAddress = logAddress;
528             _log = FileHelpers.Append(logAddress);
529             _transitions = new Queue<Transition>();
530             _transitionsPusher = new Task(TransitionsPusher);
531             _transitionsPusher.Start();
532         }
533 
534         /// <summary>
535         /// <para>
536         /// Gets the link value using the specified link.
537         /// </para>
538         /// <para></para>
539         /// </summary>
540         /// <param name="link">
541         /// <para>The link.</para>
542         /// <para></para>
543         /// </param>
544         /// <returns>
545         /// <para>A list of TLinkAddress</para>
```

```
546         /// <para></para>
547         /// </returns>
548         [MethodImpl(MethodImplOptions.AggressiveInlining)]
549         public IList<TLinkAddress> GetLinkValue(TLinkAddress link) => _links.GetLink(link);
550 
551         /// <summary>
552         /// <para>
553         /// Creates the substitution.
554         /// </para>
555         /// <para></para>
556         /// </summary>
557         /// <param name="substitution">
558         /// <para>The substitution.</para>
559         /// <para></para>
560         /// </param>
561         /// <returns>
562         /// <para>The created link index.</para>
563         /// <para></para>
564         /// </returns>
565         [MethodImpl(MethodImplOptions.AggressiveInlining)]
566         public override TLinkAddress Create(IList<TLinkAddress>? substitution,
            ↪ WriteHandler<TLinkAddress>? handler)
567         {
568             return _links.Create(new Link<TLinkAddress>(), (before, after) =>
569             {
570                 CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                    ↪  new Link<TLinkAddress>(before), new Link<TLinkAddress>(after)));
571                 return handler?.Invoke(before, after) ?? Links.Constants.Continue;
572             });
573         }
574 
575         /// <summary>
576         /// <para>
577         /// Updates the substitution.
578         /// </para>
579         /// <para></para>
580         /// </summary>
581         /// <param name="restriction">
582         /// <para>The substitution.</para>
583         /// <para></para>
584         /// </param>
585         /// <param name="substitution">
586         /// <para>The substitution.</para>
587         /// <para></para>
588         /// </param>
589         /// <returns>
590         /// <para>The link index.</para>
591         /// <para></para>
592         /// </returns>
593         [MethodImpl(MethodImplOptions.AggressiveInlining)]
594         public override TLinkAddress Update(IList<TLinkAddress>? restriction,
            ↪ IList<TLinkAddress>? substitution, WriteHandler<TLinkAddress>? handler)
595         {
596             return _links.Update(restriction, substitution, (before, after) =>
597                 {
598                     CommitTransition(new Transition(_uniqueTimestampFactory,
                        ↪  _currentTransactionId, new Link<TLinkAddress>(before), new
                        ↪  Link<TLinkAddress>(after)));
599                     return handler != null ? handler(before, after) : Constants.Continue;
600                 }
601             );
602         }
603 
604         /// <summary>
605         /// <para>
606         /// Deletes the substitution.
607         /// </para>
608         /// <para></para>
609         /// </summary>
610         /// <param name="restriction">
611         /// <para>The substitution.</para>
612         /// <para></para>
613         /// </param>
614         [MethodImpl(MethodImplOptions.AggressiveInlining)]
615         public override TLinkAddress Delete(IList<TLinkAddress>? restriction,
            ↪ WriteHandler<TLinkAddress>? handler)
616         {
```

```
617             var link = this.GetIndex(restriction);
618             return _links.Delete(restriction, (before, after) =>
619             {
620                 CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                    ↪  before, after));
621                 return handler != null ? handler(before, after) : Constants.Continue;
622             });
623         }
624         [MethodImpl(MethodImplOptions.AggressiveInlining)]
625         private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
            ↪  _transitions;
626         [MethodImpl(MethodImplOptions.AggressiveInlining)]
627         private void CommitTransition(Transition transition)
628         {
629             if (_currentTransaction != null)
630             {
631                 Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
632             }
633             var transitions = GetCurrentTransitions();
634             transitions.Enqueue(transition);
635         }
636         [MethodImpl(MethodImplOptions.AggressiveInlining)]
637         private void RevertTransition(Transition transition)
638         {
639             if (transition.After.IsNull()) // Revert Deletion with Creation
640             {
641                 _links.Create();
642             }
643             else if (transition.Before.IsNull()) // Revert Creation with Deletion
644             {
645                 _links.Delete(transition.After.Index);
646             }
647             else // Revert Update
648             {
649                 _links.Update(new[] { transition.After.Index, transition.Before.Source,
                    ↪  transition.Before.Target });
650             }
651         }
652         [MethodImpl(MethodImplOptions.AggressiveInlining)]
653         private void ResetCurrentTransation()
654         {
655             _currentTransactionId = 0;
656             _currentTransactionTransitions = null;
657             _currentTransaction = null;
658         }
659         [MethodImpl(MethodImplOptions.AggressiveInlining)]
660         private void PushTransitions()
661         {
662             if (_log == null || _transitions == null)
663             {
664                 return;
665             }
666             for (var i = 0; i < _transitions.Count; i++)
667             {
668                 var transition = _transitions.Dequeue();
669
670                 _log.Write(transition);
671                 _lastCommitedTransition = transition;
672             }
673         }
674         [MethodImpl(MethodImplOptions.AggressiveInlining)]
675         private void TransitionsPusher()
676         {
677             while (!Disposable.IsDisposed && _transitionsPusher != null)
678             {
679                 Thread.Sleep(DefaultPushDelay);
680                 PushTransitions();
681             }
682         }
683
684         /// <summary>
685         /// <para>
686         /// Begins the transaction.
687         /// </para>
688         /// <para></para>
689         /// </summary>
690         /// <returns>
691         /// <para>The transaction</para>
```

```
692         /// <para></para>
693         /// </returns>
694         [MethodImpl(MethodImplOptions.AggressiveInlining)]
695         public Transaction BeginTransaction() => new Transaction(this);
696         [MethodImpl(MethodImplOptions.AggressiveInlining)]
697         private void DisposeTransitions()
698         {
699             try
700             {
701                 var pusher = _transitionsPusher;
702                 if (pusher != null)
703                 {
704                     _transitionsPusher = null;
705                     pusher.Wait();
706                 }
707                 if (_transitions != null)
708                 {
709                     PushTransitions();
710                 }
711                 _log.DisposeIfPossible();
712                 FileHelpers.WriteFirst(_logAddress, _lastCommitedTransition);
713             }
714             catch (Exception ex)
715             {
716                 ex.Ignore();
717             }
718         }
719 
720         #region DisposalBase
721 
722         /// <summary>
723         /// <para>
724         /// Disposes the manual.
725         /// </para>
726         /// <para></para>
727         /// </summary>
728         /// <param name="manual">
729         /// <para>The manual.</para>
730         /// <para></para>
731         /// </param>
732         /// <param name="wasDisposed">
733         /// <para>The was disposed.</para>
734         /// <para></para>
735         /// </param>
736         [MethodImpl(MethodImplOptions.AggressiveInlining)]
737         protected override void Dispose(bool manual, bool wasDisposed)
738         {
739             if (!wasDisposed)
740             {
741                 DisposeTransitions();
742             }
743             base.Dispose(manual, wasDisposed);
744         }
745 
746         #endregion
747     }
748 }
```

1.120 ./csharp/Platform.Data.Doublets.Tests/GenericLinksTests.cs

```
1  using System;
2  using System.IO;
3  using Platform.Data.Doublets.Decorators;
4  using Xunit;
5 
6  using Platform.Memory;
7 
8  using Platform.Data.Doublets.Memory.United.Generic;
9 
10 namespace Platform.Data.Doublets.Tests
11 {
12     public static class GenericLinksTests
13     {
14         [Fact]
15         public static void CRUDTest()
16         {
17             Using<byte>(links => links.TestCRUDOperations());
18             Using<ushort>(links => links.TestCRUDOperations());
19             Using<uint>(links => links.TestCRUDOperations());
20             Using<ulong>(links => links.TestCRUDOperations());
```

```
21          }
22
23          [Fact]
24          public static void RawNumbersCRUDTest()
25          {
26              Using<byte>(links => links.TestRawNumbersCRUDOperations());
27              Using<ushort>(links => links.TestRawNumbersCRUDOperations());
28              Using<uint>(links => links.TestRawNumbersCRUDOperations());
29              Using<ulong>(links => links.TestRawNumbersCRUDOperations());
30          }
31
32          [Fact]
33          public static void MultipleRandomCreationsAndDeletionsTest()
34          {
35              Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
                ↪ MultipleRandomCreationsAndDeletions(16)); // Cannot use more because current
                ↪ implementation of tree cuts out 5 bits from the address space.
36              Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Te
                ↪ stMultipleRandomCreationsAndDeletions(100));
37              Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
                ↪ MultipleRandomCreationsAndDeletions(100));
38              Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
                ↪ tMultipleRandomCreationsAndDeletions(100));
39          }
40          private static void Using<TLinkAddress>(Action<ILinks<TLinkAddress>> action)
41          {
42              var unitedMemoryLinks = new UnitedMemoryLinks<TLinkAddress>(new
                ↪ HeapResizableDirectMemory());
43              using (var logFile = File.Open("linksLogger.txt", FileMode.Create, FileAccess.Write))
44              {
45                  LoggingDecorator<TLinkAddress> links = new(unitedMemoryLinks, logFile);
46                  action(links);
47              }
48
49              File.Delete("db.links");
50              using var ffiLinks = new FFI.UnitedMemoryLinks<TLinkAddress>("db.links");
51              action(ffiLinks);
52          }
53      }
54  }
```

1.121 ./csharp/Platform.Data.Doublets.Tests/ILinksBasicTests.cs

```
1   using System.IO;
2   using Platform.Data.Doublets.Memory.United.Generic;
3   using Platform.Memory;
4   using Xunit;
5
6
7   namespace Platform.Data.Doublets.Tests
8   {
9       public static class ILinksBasicTests
10      {
11          [Fact]
12          public static void DeleteAllUsages()
13          {
14              var mem = new HeapResizableDirectMemory();
15              var links = new UnitedMemoryLinks<uint>(mem);
16
17              var root = links.CreatePoint();
18
19              var a = links.CreatePoint();
20              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
```

```
            var root = links.CreatePoint();

            var a = links.CreatePoint();
            var b = links.CreatePoint();

            links.CreateAndUpdate(a, root);
            links.CreateAndUpdate(b, root);

            Assert.Equal(5U, links.Count());

            links.DeleteAllUsages(root);

            Assert.Equal(3U, links.Count());
        }
    }
}
```

1.122 ./csharp/Platform.Data.Doublets.Tests/LinksConstantsTests.cs

```
using Xunit;

namespace Platform.Data.Doublets.Tests
{
    public static class LinksConstantsTests
    {
        [Fact]
        public static void ExternalReferencesTest()
        {
            LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue), (long.MaxValue + 1UL, ulong.MaxValue));

            //var minimum = new Hybrid<ulong>(0, isExternal: true);
            var minimum = new Hybrid<ulong>(1, isExternal: true);
            var maximum = new Hybrid<ulong>(long.MaxValue, isExternal: true);

            Assert.True(constants.IsExternalReference(minimum));
            Assert.True(constants.IsExternalReference(maximum));
        }
    }
}
```

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

```
using System.IO;
using Xunit;
using Platform.Singletons;
using Platform.Memory;
using Platform.Data.Doublets.Memory.United.Specific;

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

        [Fact]
        public static void BasicFileMappedMemoryTest()
        {
            var tempFilename = Path.GetTempFileName();
            using (var memoryAdapter = new UInt64UnitedMemoryLinks(tempFilename))
            {
                memoryAdapter.TestBasicMemoryOperations();
            }
            File.Delete(tempFilename);
        }

        [Fact]
        public static void BasicHeapMemoryTest()
        {
            using (var memory = new HeapResizableDirectMemory(UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
            using (var memoryAdapter = new UInt64UnitedMemoryLinks(memory, UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
            {
                memoryAdapter.TestBasicMemoryOperations();
            }
        }
        private static void TestBasicMemoryOperations(this ILinks<ulong> memoryAdapter)
        {
            var link = memoryAdapter.Create();
```

```
            memoryAdapter.Delete(link);
        }

        [Fact]
        public static void NonexistentReferencesHeapMemoryTest()
        {
            using (var memory = new
            ↪  HeapResizableDirectMemory(UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
            using (var memoryAdapter = new UInt64UnitedMemoryLinks(memory,
            ↪  UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
            {
                memoryAdapter.TestNonexistentReferences();
            }
        }
        private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
        {
            var link = memoryAdapter.Create();
            memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
            var resultLink = _constants.Null;
            memoryAdapter.Each(foundLink =>
            {
                resultLink = memoryAdapter.GetIndex(foundLink);
                return _constants.Break;
            }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
            Assert.True(resultLink == link);
            Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
            memoryAdapter.Delete(link);
        }
    }
}
```

1.124 ./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()
```

```
48          {
49              using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪ UnitedMemoryLinks<ulong>>>())
50              {
51                  var links = scope.Use<ILinks<ulong>>();
52                  Assert.IsType<UnitedMemoryLinks<ulong>>(links);
53              }
54          }
55      }
56  }
```

1.125 ./csharp/Platform.Data.Doublets.Tests/SplitMemoryGenericLinksTests.cs

```
1   using System;
2   using Xunit;
3   using Platform.Memory;
4   using Platform.Data.Doublets.Memory.Split.Generic;
5   using Platform.Data.Doublets.Memory;
6
7   namespace Platform.Data.Doublets.Tests
8   {
9       public unsafe static class SplitMemoryGenericLinksTests
10      {
11          [Fact]
12          public static void CRUDTest()
13          {
14              Using<byte>(links => links.TestCRUDOperations());
15              Using<ushort>(links => links.TestCRUDOperations());
16              Using<uint>(links => links.TestCRUDOperations());
17              Using<ulong>(links => links.TestCRUDOperations());
18          }
19
20          [Fact]
21          public static void RawNumbersCRUDTest()
22          {
23              UsingWithExternalReferences<byte>(links => links.TestRawNumbersCRUDOperations());
24              UsingWithExternalReferences<ushort>(links => links.TestRawNumbersCRUDOperations());
25              UsingWithExternalReferences<uint>(links => links.TestRawNumbersCRUDOperations());
26              UsingWithExternalReferences<ulong>(links => links.TestRawNumbersCRUDOperations());
27          }
28
29          [Fact]
30          public static void MultipleRandomCreationsAndDeletionsTest()
31          {
32              Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
                ↪ MultipleRandomCreationsAndDeletions(16)); // Cannot use more because current
                ↪ implementation of tree cuts out 5 bits from the address space.
33              Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Te
                ↪ stMultipleRandomCreationsAndDeletions(100));
34              Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
                ↪ MultipleRandomCreationsAndDeletions(100));
35              Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
                ↪ tMultipleRandomCreationsAndDeletions(100));
36          }
37          private static void Using<TLinkAddress>(Action<ILinks<TLinkAddress>> action)
38          {
39              using (var dataMemory = new HeapResizableDirectMemory())
40              using (var indexMemory = new HeapResizableDirectMemory())
41              using (var memory = new SplitMemoryLinks<TLinkAddress>(dataMemory, indexMemory))
42              {
43                  action(memory);
44              }
45          }
46          private static void
                ↪ UsingWithExternalReferences<TLinkAddress>(Action<ILinks<TLinkAddress>> action)
47          {
48              var contants = new LinksConstants<TLinkAddress>(enableExternalReferencesSupport:
                ↪ true);
49              using (var dataMemory = new HeapResizableDirectMemory())
50              using (var indexMemory = new HeapResizableDirectMemory())
51              using (var memory = new SplitMemoryLinks<TLinkAddress>(dataMemory, indexMemory,
                ↪ SplitMemoryLinks<TLinkAddress>.DefaultLinksSizeStep, contants))
52              {
53                  action(memory);
54              }
55          }
56      }
57  }
```

1.126 ./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);
            }
        }
    }
}
```

1.127 ./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);
            }
        }
    }
}
```

1.128 ./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;
using Platform.Delegates;

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,T>(constants.Continue, constants.Break, constants.Null);
            links.Each(new Link<T>(constants.Any, constants.Any, constants.Any),
            ↪  setter.SetFirstFromListAndReturnTrue);

            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,T>(constants.Continue, constants.Break, constants.Null);
            links.Each(new Link<T>(constants.Any, constants.Any, constants.Any),
            ↪  setter.SetFirstFromListAndReturnTrue);

            Assert.True(equalityComparer.Equals(setter.Result, linkAddress));

            // Update link to reference itself
            links.Update(linkAddress, linkAddress, linkAddress);
```

```
50
51              link = new Link<T>(links.GetLink(linkAddress));
52
53              Assert.True(equalityComparer.Equals(link.Source, linkAddress));
54              Assert.True(equalityComparer.Equals(link.Target, linkAddress));
55
56              // Update link to reference null (prepare for delete)
57              var updated = links.Update(linkAddress, constants.Null, constants.Null);
58
59              Assert.True(equalityComparer.Equals(updated, linkAddress));
60
61              link = new Link<T>(links.GetLink(linkAddress));
62
63              Assert.True(equalityComparer.Equals(link.Source, constants.Null));
64              Assert.True(equalityComparer.Equals(link.Target, constants.Null));
65
66              // Delete link
67              links.Delete(linkAddress);
68
69              Assert.True(equalityComparer.Equals(links.Count(), zero));
70
71              setter = new Setter<T,T>(constants.Continue, constants.Break, constants.Null);
72              links.Each(new Link<T>(constants.Any, constants.Any, constants.Any),
                ↪  setter.SetFirstFromListAndReturnTrue);
73
74              Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
75          }
76
77          public static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
78          {
79              // Constants
80              var constants = links.Constants;
81              var equalityComparer = EqualityComparer<T>.Default;
82
83              var zero = default(T);
84              var one = Arithmetic.Increment(zero);
85              var two = Arithmetic.Increment(one);
86
87              var h106E = new Hybrid<T>(106L, isExternal: true);
88              var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
89              var h108E = new Hybrid<T>(-108L);
90
91              Assert.Equal(106L, h106E.AbsoluteValue);
92              Assert.Equal(107L, h107E.AbsoluteValue);
93              Assert.Equal(108L, h108E.AbsoluteValue);
94
95              // Create Link (External -> External)
96              var linkAddress1 = links.Create();
97
98              links.Update(linkAddress1, h106E, h108E);
99
100             var link1 = new Link<T>(links.GetLink(linkAddress1));
101
102             Assert.True(equalityComparer.Equals(link1.Source, h106E));
103             Assert.True(equalityComparer.Equals(link1.Target, h108E));
104
105             // Create Link (Internal -> External)
106             var linkAddress2 = links.Create();
107
108             links.Update(linkAddress2, linkAddress1, h108E);
109
110             var link2 = new Link<T>(links.GetLink(linkAddress2));
111
112             Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
113             Assert.True(equalityComparer.Equals(link2.Target, h108E));
114
115             // Create Link (Internal -> Internal)
116             var linkAddress3 = links.Create();
117
118             links.Update(linkAddress3, linkAddress1, linkAddress2);
119
120             var link3 = new Link<T>(links.GetLink(linkAddress3));
121
122             Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
123             Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));
124
125             // Search for created link
126             var setter1 = new Setter<T, T>(constants.Continue, constants.Break, constants.Null);
127             links.Each(new Link<T>(constants.Any, h106E, h108E),
                ↪  setter1.SetFirstFromListAndReturnTrue);
```

```
128
129             Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));
130
131             // Search for nonexistent link
132             var setter2 = new Setter<T, T>(constants.Continue, constants.Break, constants.Null);
133             links.Each(new Link<T>(constants.Any, h106E, h108E),
                ↪  setter1.SetFirstFromListAndReturnTrue);
134
135             Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));
136
137             // Update link to reference null (prepare for delete)
138             var updated = links.Update(linkAddress3, constants.Null, constants.Null);
139
140             Assert.True(equalityComparer.Equals(updated, linkAddress3));
141
142             link3 = new Link<T>(links.GetLink(linkAddress3));
143
144             Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
145             Assert.True(equalityComparer.Equals(link3.Target, constants.Null));
146
147             // Delete link
148             links.Delete(linkAddress3);
149
150             Assert.True(equalityComparer.Equals(links.Count(), two));
151
152             var setter3 = new Setter<T, T>(constants.Continue, constants.Break, constants.Null);
153             links.Each(new Link<T>(constants.Any, constants.Any, constants.Any),
                ↪  setter3.SetFirstFromListAndReturnTrue);
154
155             Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
156         }
157
158         public static void TestMultipleCreationsAndDeletions<TLinkAddress>(this
            ↪  ILinks<TLinkAddress> links, int numberOfOperations)
159         {
160             for (int i = 0; i < numberOfOperations; i++)
161             {
162                 links.Create();
163             }
164             for (int i = 0; i < numberOfOperations; i++)
165             {
166                 links.Delete(links.Count());
167             }
168         }
169
170         public static void TestMultipleRandomCreationsAndDeletions<TLinkAddress>(this
            ↪  ILinks<TLinkAddress> links, int maximumOperationsPerCycle)
171         {
172             var comparer = Comparer<TLinkAddress>.Default;
173             var addressToUInt64Converter = CheckedConverter<TLinkAddress, ulong>.Default;
174             var uInt64ToAddressConverter = CheckedConverter<ulong, TLinkAddress>.Default;
175             for (var N = 1; N < maximumOperationsPerCycle; N++)
176             {
177                 var random = new System.Random(N);
178                 var created = 0UL;
179                 var deleted = 0UL;
180                 for (var i = 0; i < N; i++)
181                 {
182                     var linksCount = addressToUInt64Converter.Convert(links.Count());
183                     var createPoint = random.NextBoolean();
184                     if (linksCount >= 2 && createPoint)
185                     {
186                         var linksAddressRange = new Range<ulong>(1, linksCount);
187                         TLinkAddress source = uInt64ToAddressConverter.Convert(random.NextUInt64
                            ↪  (linksAddressRange));
188                         TLinkAddress target = uInt64ToAddressConverter.Convert(random.NextUInt64
                            ↪  (linksAddressRange));
                            ↪  //-V3086
189                         var resultLink = links.GetOrCreate(source, target);
190                         if (comparer.Compare(resultLink,
                            ↪  uInt64ToAddressConverter.Convert(linksCount)) > 0)
191                         {
192                             created++;
193                         }
194                     }
195                     else
196                     {
197                         links.Create();
198                         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);
            }
        }
    }
}
```

1.129 ./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)
        {
            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);
        }
    }
}
```

1.130 ./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>>());
            }
        }
    }
}
```

1.131 ./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