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

        private readonly TLink _targetToMatch;

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

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

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

```
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

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

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

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

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

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

        protected ILinks<TLink> _facade;

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

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

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

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

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

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

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

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

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

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

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

            protected override bool AllowMultipleDisposeCalls
```

```
            {
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                get => true;
            }
        }

        protected readonly DisposableWithMultipleCallsAllowed Disposable;

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
14          public override TLink Create(IList<TLink> restrictions) => _links.CreatePoint();
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
                ↪ _links.Update(restrictions, _links.ResolveConstantAsSelfReference(_constants.Null,
                ↪ restrictions, substitution));
18      }
19  }
```

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

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Decorators
7   {
8       public class LinksUniquenessResolver<TLink> : LinksDecoratorBase<TLink>
9       {
10          private static readonly EqualityComparer<TLink> _equalityComparer =
                ↪ EqualityComparer<TLink>.Default;
11
12          [MethodImpl(MethodImplOptions.AggressiveInlining)]
13          public LinksUniquenessResolver(ILinks<TLink> links) : base(links) { }
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
17          {
18              var constants = _constants;
19              var links = _links;
20              var newLinkAddress = links.SearchOrDefault(substitution[constants.SourcePart],
                    ↪ substitution[constants.TargetPart]);
21              if (_equalityComparer.Equals(newLinkAddress, default))
22              {
23                  return links.Update(restrictions, substitution);
24              }
25              return ResolveAddressChangeConflict(restrictions[constants.IndexPart],
                    ↪ newLinkAddress);
26          }
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          protected virtual TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
                ↪ newLinkAddress)
30          {
31              if (!_equalityComparer.Equals(oldLinkAddress, newLinkAddress) &&
                    ↪ _links.Exists(oldLinkAddress))
32              {
33                  _facade.Delete(oldLinkAddress);
34              }
35              return newLinkAddress;
36          }
37      }
38  }
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using TLink = System.UInt32;

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

namespace Platform.Data.Doublets.Decorators
{
    public class UInt32Links : LinksDisposableDecoratorBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32Links(ILinks<TLink> links) : base(links) { }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
        {
            var constants = _constants;
            var indexPartConstant = constants.IndexPart;
            var sourcePartConstant = constants.SourcePart;
            var targetPartConstant = constants.TargetPart;
```

```
            var nullConstant = constants.Null;
            var itselfConstant = constants.Itself;
            var existedLink = nullConstant;
            var updatedLink = restrictions[indexPartConstant];
            var newSource = substitution[sourcePartConstant];
            var newTarget = substitution[targetPartConstant];
            var links = _links;
            if (newSource != itselfConstant && newTarget != itselfConstant)
            {
                existedLink = links.SearchOrDefault(newSource, newTarget);
            }
            if (existedLink == nullConstant)
            {
                var before = links.GetLink(updatedLink);
                if (before[sourcePartConstant] != newSource || before[targetPartConstant] !=
                    newTarget)
                {
                    links.Update(updatedLink, newSource == itselfConstant ? updatedLink :
                        newSource,
                                                newTarget == itselfConstant ? updatedLink :
                                                    newTarget);
                }
                return updatedLink;
            }
            else
            {
                return _facade.MergeAndDelete(updatedLink, existedLink);
            }
        }

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

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

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

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

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

```
28      {
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          public UInt64Links(ILinks<ulong> links) : base(links) { }
31
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          public override ulong Create(IList<ulong> restrictions) => _links.CreatePoint();
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
37          {
38              var constants = _constants;
39              var indexPartConstant = constants.IndexPart;
40              var sourcePartConstant = constants.SourcePart;
41              var targetPartConstant = constants.TargetPart;
42              var nullConstant = constants.Null;
43              var itselfConstant = constants.Itself;
44              var existedLink = nullConstant;
45              var updatedLink = restrictions[indexPartConstant];
46              var newSource = substitution[sourcePartConstant];
47              var newTarget = substitution[targetPartConstant];
48              var links = _links;
49              if (newSource != itselfConstant && newTarget != itselfConstant)
50              {
51                  existedLink = links.SearchOrDefault(newSource, newTarget);
52              }
53              if (existedLink == nullConstant)
54              {
55                  var before = links.GetLink(updatedLink);
56                  if (before[sourcePartConstant] != newSource || before[targetPartConstant] !=
                        ↪ newTarget)
57                  {
58                      links.Update(updatedLink, newSource == itselfConstant ? updatedLink :
                            ↪ newSource,
59                                                newTarget == itselfConstant ? updatedLink :
                                                  ↪ newTarget);
60                  }
61                  return updatedLink;
62              }
63              else
64              {
65                  return _facade.MergeAndDelete(updatedLink, existedLink);
66              }
67          }
68
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          public override void Delete(IList<ulong> restrictions)
71          {
72              var linkIndex = restrictions[_constants.IndexPart];
73              var links = _links;
74              links.EnforceResetValues(linkIndex);
75              _facade.DeleteAllUsages(linkIndex);
76              links.Delete(linkIndex);
77          }
78      }
79  }
```

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

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using Platform.Collections;
5   using Platform.Collections.Lists;
6   using Platform.Data.Universal;
7
8   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10  namespace Platform.Data.Doublets.Decorators
11  {
12      /// <remarks>
13      /// What does empty pattern (for condition or substitution) mean? Nothing or Everything?
14      /// 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).
15      ///
16      /// 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.)
17      /// </remarks>
18      internal class UniLinks<TLink> : LinksDecoratorBase<TLink>, IUniLinks<TLink>
19      {
```

```
20          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪ EqualityComparer<TLink>.Default;
21
22          public UniLinks(ILinks<TLink> links) : base(links) { }
23
24          private struct Transition
25          {
26              public IList<TLink> Before;
27              public IList<TLink> After;
28
29              public Transition(IList<TLink> before, IList<TLink> after)
30              {
31                  Before = before;
32                  After = after;
33              }
34          }
35
36          //public static readonly TLink NullConstant = Use<LinksConstants<TLink>>.Single.Null;
37          //public static readonly IReadOnlyList<TLink> NullLink = new
            ↪ ReadOnlyCollection<TLink>(new List<TLink> { NullConstant, NullConstant, NullConstant
            ↪ });
38
39          // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution
            ↪ (Links-Expression)
40          public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>, TLink>
            ↪ matchedHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
            ↪ substitutedHandler)
41          {
42              ////List<Transition> transitions = null;
43              ////if (!restriction.IsNullOrEmpty())
44              ////{
45              ////    // Есть причина делать проход (чтение)
46              ////    if (matchedHandler != null)
47              ////    {
48              ////        if (!substitution.IsNullOrEmpty())
49              ////        {
50              ////            // restriction => { 0, 0, 0 } | { 0 } // Create
51              ////            // substitution => { itself, 0, 0 } | { itself, itself, itself } //
                ↪ Create / Update
52              ////            // substitution => { 0, 0, 0 } | { 0 } // Delete
53              ////            transitions = new List<Transition>();
54              ////            if (Equals(substitution[Constants.IndexPart], Constants.Null))
55              ////            {
56              ////                // If index is Null, that means we always ignore every other
                ↪ value (they are also Null by definition)
57              ////                var matchDecision = matchedHandler(, NullLink);
58              ////                if (Equals(matchDecision, Constants.Break))
59              ////                    return false;
60              ////                if (!Equals(matchDecision, Constants.Skip))
61              ////                    transitions.Add(new Transition(matchedLink, newValue));
62              ////            }
63              ////            else
64              ////            {
65              ////                Func<T, bool> handler;
66              ////                handler = link =>
67              ////                {
68              ////                    var matchedLink = Memory.GetLinkValue(link);
69              ////                    var newValue = Memory.GetLinkValue(link);
70              ////                    newValue[Constants.IndexPart] = Constants.Itself;
71              ////                    newValue[Constants.SourcePart] =
                ↪ Equals(substitution[Constants.SourcePart], Constants.Itself) ?
                ↪ matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
72              ////                    newValue[Constants.TargetPart] =
                ↪ Equals(substitution[Constants.TargetPart], Constants.Itself) ?
                ↪ matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
73              ////                    var matchDecision = matchedHandler(matchedLink, newValue);
74              ////                    if (Equals(matchDecision, Constants.Break))
75              ////                        return false;
76              ////                    if (!Equals(matchDecision, Constants.Skip))
77              ////                        transitions.Add(new Transition(matchedLink, newValue));
78              ////                    return true;
79              ////                };
80              ////                if (!Memory.Each(handler, restriction))
81              ////                    return Constants.Break;
82              ////            }
83              ////        }
84              ////        else
85              ////        {
```

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

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

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

```
                    return substitutionHandler(before, after);
                }
                return constants.Continue;
            }
            else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
            {
                if (patternOrCondition.Count == 1)
                {
                    var linkToDelete = patternOrCondition[0];
                    var before = _links.GetLink(linkToDelete);
                    if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                    ↪  constants.Break))
                    {
                        return constants.Break;
                    }
                    var after = Array.Empty<TLink>();
                    _links.Update(linkToDelete, constants.Null, constants.Null);
                    _links.Delete(linkToDelete);
                    if (matchHandler != null)
                    {
                        return substitutionHandler(before, after);
                    }
                    return constants.Continue;
                }
                else
                {
                    throw new NotSupportedException();
                }
            }
            else // Replace / Update
            {
                if (patternOrCondition.Count == 1) //-V3125
                {
                    var linkToUpdate = patternOrCondition[0];
                    var before = _links.GetLink(linkToUpdate);
                    if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                    ↪  constants.Break))
                    {
                        return constants.Break;
                    }
                    var after = (IList<TLink>)substitution.ToArray(); //-V3125
                    if (_equalityComparer.Equals(after[0], default))
                    {
                        after[0] = linkToUpdate;
                    }
                    if (substitution.Count == 1)
                    {
                        if (!_equalityComparer.Equals(substitution[0], linkToUpdate))
                        {
                            after = _links.GetLink(substitution[0]);
                            _links.Update(linkToUpdate, constants.Null, constants.Null);
                            _links.Delete(linkToUpdate);
                        }
                    }
                    else if (substitution.Count == 3)
                    {
                        //Links.Update(after);
                    }
                    else
                    {
                        throw new NotSupportedException();
                    }
                    if (matchHandler != null)
                    {
                        return substitutionHandler(before, after);
                    }
                    return constants.Continue;
                }
                else
                {
                    throw new NotSupportedException();
                }
            }
        }

        /// <remarks>
        /// IList[IList[IList[T]]]
        /// |      |      |      |||
```

```
309             /// |      |      ------ ||
310             /// |      |       link  ||
311             /// |      -------------- |
312             /// |         change      |
313             ///  ---------------------
314             ///        changes
315             /// </remarks>
316             public IList<IList<IList<TLink>>> Trigger(IList<TLink> condition, IList<TLink>
                ↪  substitution)
317             {
318                 var changes = new List<IList<IList<TLink>>>();
319                 var @continue = _constants.Continue;
320                 Trigger(condition, AlwaysContinue, substitution, (before, after) =>
321                 {
322                     var change = new[] { before, after };
323                     changes.Add(change);
324                     return @continue;
325                 });
326                 return changes;
327             }
328
329             private TLink AlwaysContinue(IList<TLink> linkToMatch) => _constants.Continue;
330         }
331     }
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
using Platform.Collections.Arrays;
using Platform.Random;
using Platform.Setters;
using Platform.Converters;
using Platform.Numbers;
using Platform.Data.Exceptions;
using Platform.Data.Doublets.Decorators;

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

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

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

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

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

```

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

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

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

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

        #region Paths

        /// <remarks>
        /// TODO: Как так? Как то что ниже может быть корректно?
        /// Скорее всего практически не применимо
        /// Предполагалось, что можно было конвертировать формируемый в проходе через
        ↪  SequenceWalker
```

```
150         /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
151         /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
152         /// </remarks>
153         [MethodImpl(MethodImplOptions.AggressiveInlining)]
154         public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[]
            ↪ path)
155         {
156             var current = path[0];
157             //EnsureLinkExists(current, "path");
158             if (!links.Exists(current))
159             {
160                 return false;
161             }
162             var equalityComparer = EqualityComparer<TLink>.Default;
163             var constants = links.Constants;
164             for (var i = 1; i < path.Length; i++)
165             {
166                 var next = path[i];
167                 var values = links.GetLink(current);
168                 var source = values[constants.SourcePart];
169                 var target = values[constants.TargetPart];
170                 if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
                    ↪ next))
171                 {
172                     //throw new InvalidOperationException(string.Format("Невозможно выбрать
                        ↪ путь, так как и Source и Target совпадают с элементом пути {0}.", next));
173                     return false;
174                 }
175                 if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
                    ↪ target))
176                 {
177                     //throw new InvalidOperationException(string.Format("Невозможно продолжить
                        ↪ путь через элемент пути {0}", next));
178                     return false;
179                 }
180                 current = next;
181             }
182             return true;
183         }
184 
185         /// <remarks>
186         /// Может потребовать дополнительного стека для PathElement's при использовании
            ↪ SequenceWalker.
187         /// </remarks>
188         [MethodImpl(MethodImplOptions.AggressiveInlining)]
189         public static TLink GetByKeys<TLink>(this ILinks<TLink> links, TLink root, params int[]
            ↪ path)
190         {
191             links.EnsureLinkExists(root, "root");
192             var currentLink = root;
193             for (var i = 0; i < path.Length; i++)
194             {
195                 currentLink = links.GetLink(currentLink)[path[i]];
196             }
197             return currentLink;
198         }
199 
200         [MethodImpl(MethodImplOptions.AggressiveInlining)]
201         public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this ILinks<TLink>
            ↪ links, TLink root, ulong size, ulong index)
202         {
203             var constants = links.Constants;
204             var source = constants.SourcePart;
205             var target = constants.TargetPart;
206             if (!Platform.Numbers.Math.IsPowerOfTwo(size))
207             {
208                 throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes other
                    ↪ than powers of two are not supported.");
209             }
210             var path = new BitArray(BitConverter.GetBytes(index));
211             var length = Bit.GetLowestPosition(size);
212             links.EnsureLinkExists(root, "root");
213             var currentLink = root;
214             for (var i = length - 1; i >= 0; i--)
215             {
216                 currentLink = links.GetLink(currentLink)[path[i] ? target : source];
217             }
218             return currentLink;
```

```
219         }
220
221         #endregion
222
223         /// <summary>
224         /// Возвращает индекс указанной связи.
225         /// </summary>
226         /// <param name="links">Хранилище связей.</param>
227         /// <param name="link">Связь представленная списком, состоящим из её адреса и
            ↪  содержимого.</param>
228         /// <returns>Индекс начальной связи для указанной связи.</returns>
229         [MethodImpl(MethodImplOptions.AggressiveInlining)]
230         public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
            ↪  link[links.Constants.IndexPart];
231
232         /// <summary>
233         /// Возвращает индекс начальной (Source) связи для указанной связи.
234         /// </summary>
235         /// <param name="links">Хранилище связей.</param>
236         /// <param name="link">Индекс связи.</param>
237         /// <returns>Индекс начальной связи для указанной связи.</returns>
238         [MethodImpl(MethodImplOptions.AggressiveInlining)]
239         public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink link) =>
            ↪  links.GetLink(link)[links.Constants.SourcePart];
240
241         /// <summary>
242         /// Возвращает индекс начальной (Source) связи для указанной связи.
243         /// </summary>
244         /// <param name="links">Хранилище связей.</param>
245         /// <param name="link">Связь представленная списком, состоящим из её адреса и
            ↪  содержимого.</param>
246         /// <returns>Индекс начальной связи для указанной связи.</returns>
247         [MethodImpl(MethodImplOptions.AggressiveInlining)]
248         public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
            ↪  link[links.Constants.SourcePart];
249
250         /// <summary>
251         /// Возвращает индекс конечной (Target) связи для указанной связи.
252         /// </summary>
253         /// <param name="links">Хранилище связей.</param>
254         /// <param name="link">Индекс связи.</param>
255         /// <returns>Индекс конечной связи для указанной связи.</returns>
256         [MethodImpl(MethodImplOptions.AggressiveInlining)]
257         public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
            ↪  links.GetLink(link)[links.Constants.TargetPart];
258
259         /// <summary>
260         /// Возвращает индекс конечной (Target) связи для указанной связи.
261         /// </summary>
262         /// <param name="links">Хранилище связей.</param>
263         /// <param name="link">Связь представленная списком, состоящим из её адреса и
            ↪  содержимого.</param>
264         /// <returns>Индекс конечной связи для указанной связи.</returns>
265         [MethodImpl(MethodImplOptions.AggressiveInlining)]
266         public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
            ↪  link[links.Constants.TargetPart];
267
268         /// <summary>
269         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪  (handler) для каждой подходящей связи.
270         /// </summary>
271         /// <param name="links">Хранилище связей.</param>
272         /// <param name="handler">Обработчик каждой подходящей связи.</param>
273         /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
            ↪  может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
            ↪  Any - отсутствие ограничения, 1..∞ конкретный адрес связи.</param>
274         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪  случае.</returns>
275         [MethodImpl(MethodImplOptions.AggressiveInlining)]
276         public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
            ↪  handler, params TLink[] restrictions)
277             => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
                ↪  links.Constants.Continue);
278
279         /// <summary>
280         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪  (handler) для каждой подходящей связи.
```

```
281         /// </summary>
282         /// <param name="links">Хранилище связей.</param>
283         /// <param name="source">Значение, определяющее соответствующие шаблону связи.
            ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
            ↪ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
284         /// <param name="target">Значение, определяющее соответствующие шаблону связи.
            ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
            ↪ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
285         /// <param name="handler">Обработчик каждой подходящей связи.</param>
286         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪ случае.</returns>
287         [MethodImpl(MethodImplOptions.AggressiveInlining)]
288         public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
            ↪ Func<TLink, bool> handler)
289         {
290             var constants = links.Constants;
291             return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
                ↪ constants.Break, constants.Any, source, target);
292         }
293
294         /// <summary>
295         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪ (handler) для каждой подходящей связи.
296         /// </summary>
297         /// <param name="links">Хранилище связей.</param>
298         /// <param name="source">Значение, определяющее соответствующие шаблону связи.
            ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
            ↪ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
299         /// <param name="target">Значение, определяющее соответствующие шаблону связи.
            ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
            ↪ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
300         /// <param name="handler">Обработчик каждой подходящей связи.</param>
301         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪ случае.</returns>
302         [MethodImpl(MethodImplOptions.AggressiveInlining)]
303         public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
            ↪ Func<IList<TLink>, TLink> handler) => links.Each(handler, links.Constants.Any,
            ↪ source, target);
304
305         [MethodImpl(MethodImplOptions.AggressiveInlining)]
306         public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
            ↪ restrictions)
307         {
308             var arraySize = CheckedConverter<TLink,
                ↪ ulong>.Default.Convert(links.Count(restrictions));
309             if (arraySize > 0)
310             {
311                 var array = new IList<TLink>[arraySize];
312                 var filler = new ArrayFiller<IList<TLink>, TLink>(array,
                    ↪ links.Constants.Continue);
313                 links.Each(filler.AddAndReturnConstant, restrictions);
314                 return array;
315             }
316             else
317             {
318                 return Array.Empty<IList<TLink>>();
319             }
320         }
321
322         [MethodImpl(MethodImplOptions.AggressiveInlining)]
323         public static IList<TLink> AllIndices<TLink>(this ILinks<TLink> links, params TLink[]
            ↪ restrictions)
324         {
325             var arraySize = CheckedConverter<TLink,
                ↪ ulong>.Default.Convert(links.Count(restrictions));
326             if (arraySize > 0)
327             {
328                 var array = new TLink[arraySize];
329                 var filler = new ArrayFiller<TLink, TLink>(array, links.Constants.Continue);
330                 links.Each(filler.AddFirstAndReturnConstant, restrictions);
331                 return array;
332             }
333             else
334             {
335                 return Array.Empty<TLink>();
336             }
337         }
```

```
338
339         /// <summary>
340         /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
            ↪ в хранилище связей.
341         /// </summary>
342         /// <param name="links">Хранилище связей.</param>
343         /// <param name="source">Начало связи.</param>
344         /// <param name="target">Конец связи.</param>
345         /// <returns>Значение, определяющее существует ли связь.</returns>
346         [MethodImpl(MethodImplOptions.AggressiveInlining)]
347         public static bool Exists<TLink>(this ILinks<TLink> links, TLink source, TLink target)
            ↪ => Comparer<TLink>.Default.Compare(links.Count(links.Constants.Any, source, target),
            ↪ default) > 0;
348
349         #region Ensure
350         // TODO: May be move to EnsureExtensions or make it both there and here
351
352         [MethodImpl(MethodImplOptions.AggressiveInlining)]
353         public static void EnsureLinkExists<TLink>(this ILinks<TLink> links, IList<TLink>
            ↪ restrictions)
354         {
355             for (var i = 0; i < restrictions.Count; i++)
356             {
357                 if (!links.Exists(restrictions[i]))
358                 {
359                     throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
                        ↪ $"sequence[{i}]");
360                 }
361             }
362         }
363
364         [MethodImpl(MethodImplOptions.AggressiveInlining)]
365         public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
            ↪ reference, string argumentName)
366         {
367             if (links.Constants.IsInternalReference(reference) && !links.Exists(reference))
368             {
369                 throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
370             }
371         }
372
373         [MethodImpl(MethodImplOptions.AggressiveInlining)]
374         public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
            ↪ IList<TLink> restrictions, string argumentName)
375         {
376             for (int i = 0; i < restrictions.Count; i++)
377             {
378                 links.EnsureInnerReferenceExists(restrictions[i], argumentName);
379             }
380         }
381
382         [MethodImpl(MethodImplOptions.AggressiveInlining)]
383         public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
            ↪ restrictions)
384         {
385             var equalityComparer = EqualityComparer<TLink>.Default;
386             var any = links.Constants.Any;
387             for (var i = 0; i < restrictions.Count; i++)
388             {
389                 if (!equalityComparer.Equals(restrictions[i], any) &&
                    ↪ !links.Exists(restrictions[i]))
390                 {
391                     throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
                        ↪ $"sequence[{i}]");
392                 }
393             }
394         }
395
396         [MethodImpl(MethodImplOptions.AggressiveInlining)]
397         public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
            ↪ string argumentName)
398         {
399             var equalityComparer = EqualityComparer<TLink>.Default;
400             if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
401             {
402                 throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
403             }
```

```
404         }
405
406         [MethodImpl(MethodImplOptions.AggressiveInlining)]
407         public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links, TLink
            ↪ link, string argumentName)
408         {
409             var equalityComparer = EqualityComparer<TLink>.Default;
410             if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
411             {
412                 throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
413             }
414         }
415
416         /// <param name="links">Хранилище связей.</param>
417         [MethodImpl(MethodImplOptions.AggressiveInlining)]
418         public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink source,
            ↪ TLink target)
419         {
420             if (links.Exists(source, target))
421             {
422                 throw new LinkWithSameValueAlreadyExistsException();
423             }
424         }
425
426         /// <param name="links">Хранилище связей.</param>
427         [MethodImpl(MethodImplOptions.AggressiveInlining)]
428         public static void EnsureNoUsages<TLink>(this ILinks<TLink> links, TLink link)
429         {
430             if (links.HasUsages(link))
431             {
432                 throw new ArgumentLinkHasDependenciesException<TLink>(link);
433             }
434         }
435
436         /// <param name="links">Хранилище связей.</param>
437         [MethodImpl(MethodImplOptions.AggressiveInlining)]
438         public static void EnsureCreated<TLink>(this ILinks<TLink> links, params TLink[]
            ↪ addresses) => links.EnsureCreated(links.Create, addresses);
439
440         /// <param name="links">Хранилище связей.</param>
441         [MethodImpl(MethodImplOptions.AggressiveInlining)]
442         public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params TLink[]
            ↪ addresses) => links.EnsureCreated(links.CreatePoint, addresses);
443
444         /// <param name="links">Хранилище связей.</param>
445         [MethodImpl(MethodImplOptions.AggressiveInlining)]
446         public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
            ↪ params TLink[] addresses)
447         {
448             var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
449             var uInt64ToAddressConverter = CheckedConverter<ulong, TLink>.Default;
450             var nonExistentAddresses = new HashSet<TLink>(addresses.Where(x =>
                ↪ !links.Exists(x)));
451             if (nonExistentAddresses.Count > 0)
452             {
453                 var max = nonExistentAddresses.Max();
454                 max = uInt64ToAddressConverter.Convert(System.Math.Min(addressToUInt64Converter.
                    ↪ Convert(max),
                    ↪ addressToUInt64Converter.Convert(links.Constants.InternalReferencesRange.Max
                    ↪ imum)));
455                 var createdLinks = new List<TLink>();
456                 var equalityComparer = EqualityComparer<TLink>.Default;
457                 TLink createdLink = creator();
458                 while (!equalityComparer.Equals(createdLink, max))
459                 {
460                     createdLinks.Add(createdLink);
461                 }
462                 for (var i = 0; i < createdLinks.Count; i++)
463                 {
464                     if (!nonExistentAddresses.Contains(createdLinks[i]))
465                     {
466                         links.Delete(createdLinks[i]);
467                     }
468                 }
469             }
470         }
471
472         #endregion
```

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

```
542             /// на связь с указанными началом (NewSource) и концом (NewTarget).
543             /// </summary>
544             /// <param name="links">Хранилище связей.</param>
545             /// <param name="link">Индекс обновляемой связи.</param>
546             /// <param name="newSource">Индекс связи, которая является началом связи, на которую
                ↪  выполняется обновление.</param>
547             /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
                ↪  выполняется обновление.</param>
548             /// <returns>Индекс обновлённой связи.</returns>
549             [MethodImpl(MethodImplOptions.AggressiveInlining)]
550             public static TLink Update<TLink>(this ILinks<TLink> links, TLink link, TLink newSource,
                ↪  TLink newTarget) => links.Update(new LinkAddress<TLink>(link), new Link<TLink>(link,
                ↪  newSource, newTarget));
551
552             /// <summary>
553             /// Обновляет связь с указанными началом (Source) и концом (Target)
554             /// на связь с указанными началом (NewSource) и концом (NewTarget).
555             /// </summary>
556             /// <param name="links">Хранилище связей.</param>
557             /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
                ↪  может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
                ↪  Itself - требование установить ссылку на себя, 1..∞ конкретный адрес другой
                ↪  связи.</param>
558             /// <returns>Индекс обновлённой связи.</returns>
559             [MethodImpl(MethodImplOptions.AggressiveInlining)]
560             public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
561             {
562                 if (restrictions.Length == 2)
563                 {
564                     return links.MergeAndDelete(restrictions[0], restrictions[1]);
565                 }
566                 if (restrictions.Length == 4)
567                 {
568                     return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
                        ↪  restrictions[2], restrictions[3]);
569                 }
570                 else
571                 {
572                     return links.Update(new LinkAddress<TLink>(restrictions[0]), restrictions);
573                 }
574             }
575
576             [MethodImpl(MethodImplOptions.AggressiveInlining)]
577             public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
                ↪  links, TLink constant, IList<TLink> restrictions, IList<TLink> substitution)
578             {
579                 var equalityComparer = EqualityComparer<TLink>.Default;
580                 var constants = links.Constants;
581                 var restrictionsIndex = restrictions[constants.IndexPart];
582                 var substitutionIndex = substitution[constants.IndexPart];
583                 if (equalityComparer.Equals(substitutionIndex, default))
584                 {
585                     substitutionIndex = restrictionsIndex;
586                 }
587                 var source = substitution[constants.SourcePart];
588                 var target = substitution[constants.TargetPart];
589                 source = equalityComparer.Equals(source, constant) ? substitutionIndex : source;
590                 target = equalityComparer.Equals(target, constant) ? substitutionIndex : target;
591                 return new Link<TLink>(substitutionIndex, source, target);
592             }
593
594             /// <summary>
595             /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
                ↪  с указанными Source (началом) и Target (концом).
596             /// </summary>
597             /// <param name="links">Хранилище связей.</param>
598             /// <param name="source">Индекс связи, которая является началом на создаваемой
                ↪  связи.</param>
599             /// <param name="target">Индекс связи, которая является концом для создаваемой
                ↪  связи.</param>
600             /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
601             [MethodImpl(MethodImplOptions.AggressiveInlining)]
602             public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
                ↪  target)
603             {
604                 var link = links.SearchOrDefault(source, target);
605                 if (EqualityComparer<TLink>.Default.Equals(link, default))
```

```
            {
                link = links.CreateAndUpdate(source, target);
            }
            return link;
        }

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

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

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

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

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

```
744                 var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(oldLinkIndex)) &&
                    ↪ usagesAsSourceCount == 1 && usagesAsTargetCount == 1;
745                 if (!isStandalonePoint)
746                 {
747                     var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
748                     if (totalUsages > 0)
749                     {
750                         var usages = ArrayPool.Allocate<TLink>(totalUsages);
751                         var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
                            ↪ links.Constants.Continue);
752                         var i = 0L;
753                         if (usagesAsSourceCount > 0)
754                         {
755                             links.Each(usagesFiller.AddFirstAndReturnConstant,
                                ↪ usagesAsSourceQuery);
756                             for (; i < usagesAsSourceCount; i++)
757                             {
758                                 var usage = usages[i];
759                                 if (!equalityComparer.Equals(usage, oldLinkIndex))
760                                 {
761                                     links.Update(usage, newLinkIndex, links.GetTarget(usage));
762                                 }
763                             }
764                         }
765                         if (usagesAsTargetCount > 0)
766                         {
767                             links.Each(usagesFiller.AddFirstAndReturnConstant,
                                ↪ usagesAsTargetQuery);
768                             for (; i < usages.Length; i++)
769                             {
770                                 var usage = usages[i];
771                                 if (!equalityComparer.Equals(usage, oldLinkIndex))
772                                 {
773                                     links.Update(usage, links.GetSource(usage), newLinkIndex);
774                                 }
775                             }
776                         }
777                         ArrayPool.Free(usages);
778                     }
779                 }
780             }
781             return newLinkIndex;
782         }
783 
784         /// <summary>
785         /// Replace one link with another (replaced link is deleted, children are updated or
            ↪ deleted).
786         /// </summary>
787         [MethodImpl(MethodImplOptions.AggressiveInlining)]
788         public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
            ↪ TLink newLinkIndex)
789         {
790             var equalityComparer = EqualityComparer<TLink>.Default;
791             if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
792             {
793                 links.MergeUsages(oldLinkIndex, newLinkIndex);
794                 links.Delete(oldLinkIndex);
795             }
796             return newLinkIndex;
797         }
798 
799         [MethodImpl(MethodImplOptions.AggressiveInlining)]
800         public static ILinks<TLink>
            ↪ DecorateWithAutomaticUniquenessAndUsagesResolution<TLink>(this ILinks<TLink> links)
801         {
802             links = new LinksCascadeUsagesResolver<TLink>(links);
803             links = new NonNullContentsLinkDeletionResolver<TLink>(links);
804             links = new LinksCascadeUniquenessAndUsagesResolver<TLink>(links);
805             return links;
806         }
807 
808         [MethodImpl(MethodImplOptions.AggressiveInlining)]
809         public static string Format<TLink>(this ILinks<TLink> links, IList<TLink> link)
810         {
811             var constants = links.Constants;
812             return $"({link[constants.IndexPart]}: {link[constants.SourcePart]}
                ↪ {link[constants.TargetPart]})";
813         }
```

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

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

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

1.22 ./csharp/Platform.Data.Doublets/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<TLink> : IEquatable<Link<TLink>>, IReadOnlyList<TLink>, IList<TLink>
18     {
19         public static readonly Link<TLink> Null = new Link<TLink>();
20
21         private static readonly LinksConstants<TLink> _constants =
           ↪ Default<LinksConstants<TLink>>.Instance;
22         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪ EqualityComparer<TLink>.Default;
23
24         private const int Length = 3;
25
26         public readonly TLink Index;
27         public readonly TLink Source;
28         public readonly TLink Target;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public Link(params TLink[] values) => SetValues(values, out Index, out Source, out
           ↪ Target);
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public Link(IList<TLink> values) => SetValues(values, out Index, out Source, out Target);
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public Link(object other)
38         {
39             if (other is Link<TLink> otherLink)
40             {
41                 SetValues(ref otherLink, out Index, out Source, out Target);
42             }
43             else if(other is IList<TLink> otherList)
44             {
45                 SetValues(otherList, out Index, out Source, out Target);
46             }
47             else
48             {
49                 throw new NotSupportedException();
50             }
51         }
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         public Link(ref Link<TLink> other) => SetValues(ref other, out Index, out Source, out
           ↪ Target);
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         public Link(TLink index, TLink source, TLink target)
```

```
        {
            Index = index;
            Source = source;
            Target = target;
        }

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

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

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

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

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

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

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

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

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

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

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

        #region IList
```

```
132
133         public int Count
134         {
135             [MethodImpl(MethodImplOptions.AggressiveInlining)]
136             get => Length;
137         }
138
139         public bool IsReadOnly
140         {
141             [MethodImpl(MethodImplOptions.AggressiveInlining)]
142             get => true;
143         }
144
145         public TLink this[int index]
146         {
147             [MethodImpl(MethodImplOptions.AggressiveInlining)]
148             get
149             {
150                 Ensure.OnDebug.ArgumentInRange(index, new Range<int>(0, Length - 1),
                    ↪  nameof(index));
151                 if (index == _constants.IndexPart)
152                 {
153                     return Index;
154                 }
155                 if (index == _constants.SourcePart)
156                 {
157                     return Source;
158                 }
159                 if (index == _constants.TargetPart)
160                 {
161                     return Target;
162                 }
163                 throw new NotSupportedException(); // Impossible path due to
                    ↪  Ensure.ArgumentInRange
164             }
165             [MethodImpl(MethodImplOptions.AggressiveInlining)]
166             set => throw new NotSupportedException();
167         }
168
169         [MethodImpl(MethodImplOptions.AggressiveInlining)]
170         IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
171
172         [MethodImpl(MethodImplOptions.AggressiveInlining)]
173         public IEnumerator<TLink> GetEnumerator()
174         {
175             yield return Index;
176             yield return Source;
177             yield return Target;
178         }
179
180         [MethodImpl(MethodImplOptions.AggressiveInlining)]
181         public void Add(TLink item) => throw new NotSupportedException();
182
183         [MethodImpl(MethodImplOptions.AggressiveInlining)]
184         public void Clear() => throw new NotSupportedException();
185
186         [MethodImpl(MethodImplOptions.AggressiveInlining)]
187         public bool Contains(TLink item) => IndexOf(item) >= 0;
188
189         [MethodImpl(MethodImplOptions.AggressiveInlining)]
190         public void CopyTo(TLink[] array, int arrayIndex)
191         {
192             Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
193             Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
                ↪  nameof(arrayIndex));
194             if (arrayIndex + Length > array.Length)
195             {
196                 throw new InvalidOperationException();
197             }
198             array[arrayIndex++] = Index;
199             array[arrayIndex++] = Source;
200             array[arrayIndex] = Target;
201         }
202
203         [MethodImpl(MethodImplOptions.AggressiveInlining)]
204         public bool Remove(TLink item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
205
206         [MethodImpl(MethodImplOptions.AggressiveInlining)]
207         public int IndexOf(TLink item)
```

```
208         {
209             if (_equalityComparer.Equals(Index, item))
210             {
211                 return _constants.IndexPart;
212             }
213             if (_equalityComparer.Equals(Source, item))
214             {
215                 return _constants.SourcePart;
216             }
217             if (_equalityComparer.Equals(Target, item))
218             {
219                 return _constants.TargetPart;
220             }
221             return -1;
222         }
223 
224         [MethodImpl(MethodImplOptions.AggressiveInlining)]
225         public void Insert(int index, TLink item) => throw new NotSupportedException();
226 
227         [MethodImpl(MethodImplOptions.AggressiveInlining)]
228         public void RemoveAt(int index) => throw new NotSupportedException();
229 
230         [MethodImpl(MethodImplOptions.AggressiveInlining)]
231         public static bool operator ==(Link<TLink> left, Link<TLink> right) =>
            ↪ left.Equals(right);
232 
233         [MethodImpl(MethodImplOptions.AggressiveInlining)]
234         public static bool operator !=(Link<TLink> left, Link<TLink> right) => !(left == right);
235 
236         #endregion
237     }
238 }
```

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

```
1  using System.Runtime.CompilerServices;
2 
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4 
5  namespace Platform.Data.Doublets
6  {
7      public static class LinkExtensions
8      {
9          [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         public static bool IsFullPoint<TLink>(this Link<TLink> link) =>
           ↪ Point<TLink>.IsFullPoint(link);
11 
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public static bool IsPartialPoint<TLink>(this Link<TLink> link) =>
           ↪ Point<TLink>.IsPartialPoint(link);
14     }
15 }
```

1.24 ./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      public abstract class LinksOperatorBase<TLink>
8      {
9          protected readonly ILinks<TLink> _links;
10 
11         public ILinks<TLink> Links
12         {
13             [MethodImpl(MethodImplOptions.AggressiveInlining)]
14             get => _links;
15         }
16 
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected LinksOperatorBase(ILinks<TLink> links) => _links = links;
19     }
20 }
```

1.25 ./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 
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

```
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          public override bool Equals(object obj) => obj is LinksHeader<TLink> linksHeader ?
            ↪  Equals(linksHeader) : false;
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          public bool Equals(LinksHeader<TLink> other)
30              => _equalityComparer.Equals(AllocatedLinks, other.AllocatedLinks)
31              && _equalityComparer.Equals(ReservedLinks, other.ReservedLinks)
32              && _equalityComparer.Equals(FreeLinks, other.FreeLinks)
33              && _equalityComparer.Equals(FirstFreeLink, other.FirstFreeLink)
34              && _equalityComparer.Equals(RootAsSource, other.RootAsSource)
35              && _equalityComparer.Equals(RootAsTarget, other.RootAsTarget)
36              && _equalityComparer.Equals(LastFreeLink, other.LastFreeLink)
37              && _equalityComparer.Equals(Reserved8, other.Reserved8);
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          public override int GetHashCode() => (AllocatedLinks, ReservedLinks, FreeLinks,
            ↪  FirstFreeLink, RootAsSource, RootAsTarget, LastFreeLink, Reserved8).GetHashCode();
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          public static bool operator ==(LinksHeader<TLink> left, LinksHeader<TLink> right) =>
            ↪  left.Equals(right);
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          public static bool operator !=(LinksHeader<TLink> left, LinksHeader<TLink> right) =>
            ↪  !(left == right);
47      }
48  }
```

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

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

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

```
        {
            var root = GetTreeRoot();
            while (!EqualToZero(root))
            {
                ref var rootLink = ref GetLinkDataPartReference(root);
                var rootSource = rootLink.Source;
                var rootTarget = rootLink.Target;
                if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                ↪  node.Key < root.Key
                {
                    root = GetLeftOrDefault(root);
                }
                else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
                ↪  node.Key > root.Key
                {
                    root = GetRightOrDefault(root);
                }
                else // node.Key == root.Key
                {
                    return root;
                }
            }
            return Zero;
        }

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

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

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

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

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

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

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

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

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

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

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

```
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          protected abstract TLink GetTreeRoot();
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          protected abstract TLink GetBasePartValue(TLink link);
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
            ↪  rootSource, TLink rootTarget);
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
            ↪  rootSource, TLink rootTarget);
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
            ↪  AsRef<LinksHeader<TLink>>(Header);
47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
            ↪  AsRef<RawLinkDataPart<TLink>>(LinksDataParts + (RawLinkDataPart<TLink>.SizeInBytes *
            ↪  _addressToInt64Converter.Convert(link)));
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected virtual ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
            ↪  ref AsRef<RawLinkIndexPart<TLink>>(LinksIndexParts +
            ↪  (RawLinkIndexPart<TLink>.SizeInBytes * _addressToInt64Converter.Convert(link)));
53
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
56          {
57              ref var link = ref GetLinkDataPartReference(linkIndex);
58              return new Link<TLink>(linkIndex, link.Source, link.Target);
59          }
60
61          [MethodImpl(MethodImplOptions.AggressiveInlining)]
62          protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
63          {
64              ref var firstLink = ref GetLinkDataPartReference(first);
65              ref var secondLink = ref GetLinkDataPartReference(second);
66              return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
67          }
68
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
71          {
72              ref var firstLink = ref GetLinkDataPartReference(first);
73              ref var secondLink = ref GetLinkDataPartReference(second);
74              return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
75          }
76
77          public TLink this[TLink index]
78          {
79              [MethodImpl(MethodImplOptions.AggressiveInlining)]
80              get
81              {
82                  var root = GetTreeRoot();
83                  if (GreaterOrEqualThan(index, GetSize(root)))
84                  {
85                      return Zero;
86                  }
87                  while (!EqualToZero(root))
88                  {
89                      var left = GetLeftOrDefault(root);
90                      var leftSize = GetSizeOrZero(left);
91                      if (LessThan(index, leftSize))
92                      {
93                          root = left;
94                          continue;
95                      }
96                      if (AreEqual(index, leftSize))
97                      {
98                          return root;
99                      }
100                     root = GetRightOrDefault(root);
```

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

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

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

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

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

```
1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Memory.Split.Generic
6  {
7      public unsafe class ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods<TLink> :
       ↪ ExternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
8      {
9          [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         public ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
           ↪ constants, byte* linksDataParts, byte* linksIndexParts, byte* header) :
           ↪ base(constants, linksDataParts, linksIndexParts, header) { }
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        protected override ref TLink GetLeftReference(TLink node) => ref
        ↪ GetLinkIndexPartReference(node).LeftAsSource;

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

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

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

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

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

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

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

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

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

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

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

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

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

```
using System.Runtime.CompilerServices;

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

```
using System.Runtime.CompilerServices;

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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

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

```
using System.Runtime.CompilerServices;

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        protected override TLink GetLeft(TLink node) =>
        ↪ GetLinkIndexPartReference(node).LeftAsTarget;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected InternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLink>
        ↪ constants, byte* linksDataParts, byte* linksIndexParts, byte* header)
```

```
        {
            LinksDataParts = linksDataParts;
            LinksIndexParts = linksIndexParts;
            Header = header;
            Break = constants.Break;
            Continue = constants.Continue;
        }

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

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

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

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

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

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

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

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

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

        /// <summary>
        /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
        ↪  (концом).
        /// </summary>
        /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
        /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
```

```
96          /// <returns>Индекс искомой связи.</returns>
97          [MethodImpl(MethodImplOptions.AggressiveInlining)]
98          public abstract TLink Search(TLink source, TLink target);
99
100         [MethodImpl(MethodImplOptions.AggressiveInlining)]
101         protected TLink SearchCore(TLink root, TLink key)
102         {
103             while (!EqualToZero(root))
104             {
105                 var rootKey = GetKeyPartValue(root);
106                 if (LessThan(key, rootKey)) // node.Key < root.Key
107                 {
108                     root = GetLeftOrDefault(root);
109                 }
110                 else if (GreaterThan(key, rootKey)) // node.Key > root.Key
111                 {
112                     root = GetRightOrDefault(root);
113                 }
114                 else // node.Key == root.Key
115                 {
116                     return root;
117                 }
118             }
119             return Zero;
120         }
121
122         // TODO: Return indices range instead of references count
123         [MethodImpl(MethodImplOptions.AggressiveInlining)]
124         public TLink CountUsages(TLink link) => GetSizeOrZero(GetTreeRoot(link));
125
126         [MethodImpl(MethodImplOptions.AggressiveInlining)]
127         public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
            ↪ EachUsageCore(@base, GetTreeRoot(@base), handler);
128
129         // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
            ↪ low-level MSIL stack.
130         [MethodImpl(MethodImplOptions.AggressiveInlining)]
131         private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
132         {
133             var @continue = Continue;
134             if (EqualToZero(link))
135             {
136                 return @continue;
137             }
138             var @break = Break;
139             if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
140             {
141                 return @break;
142             }
143             if (AreEqual(handler(GetLinkValues(link)), @break))
144             {
145                 return @break;
146             }
147             if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
148             {
149                 return @break;
150             }
151             return @continue;
152         }
153
154         [MethodImpl(MethodImplOptions.AggressiveInlining)]
155         protected override void PrintNodeValue(TLink node, StringBuilder sb)
156         {
157             ref var link = ref GetLinkDataPartReference(node);
158             sb.Append(' ');
159             sb.Append(link.Source);
160             sb.Append('-');
161             sb.Append('>');
162             sb.Append(link.Target);
163         }
164     }
165 }
```

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

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

```
using Platform.Converters;
using static System.Runtime.CompilerServices.Unsafe;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
        }

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

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

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

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

namespace Platform.Data.Doublets.Memory.Split.Generic
{
    public unsafe class InternalLinksSourcesLinkedListMethods<TLink> :
    ↪ RelativeCircularDoublyLinkedListMethods<TLink>
    {
        private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
        ↪ UncheckedConverter<TLink, long>.Default;
        private readonly byte* _linksDataParts;
        private readonly byte* _linksIndexParts;
        protected readonly TLink Break;
        protected readonly TLink Continue;

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        protected override TLink GetNext(TLink element) =>
        ↪  GetLinkIndexPartReference(element).RightAsSource;

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

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

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

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

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

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

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

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

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

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

```
using System.Runtime.CompilerServices;

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

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

```
    public unsafe class InternalLinksSourcesRecursionlessSizeBalancedTreeMethods<TLink> : InternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink> constants, byte* linksDataParts, byte* linksIndexParts, byte* header) : base(constants, linksDataParts, linksIndexParts, header) { }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Memory.Split.Generic
{
    public unsafe class InternalLinksSourcesSizeBalancedTreeMethods<TLink> : InternalLinksSizeBalancedTreeMethodsBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        public InternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink> constants,
        ↪  byte* linksDataParts, byte* linksIndexParts, byte* header) : base(constants,
        ↪  linksDataParts, linksIndexParts, header) { }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
using System.Runtime.CompilerServices;

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        protected override ref TLink GetLeftReference(TLink node) => ref
        ↪ GetLinkIndexPartReference(node).LeftAsTarget;

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

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

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

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

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

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

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

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

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

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

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

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

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

```
using System.Runtime.CompilerServices;

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        protected override ref TLink GetRightReference(TLink node) => ref
        ↪ GetLinkIndexPartReference(node).RightAsTarget;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

```
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪  indexMemory) : this(dataMemory, indexMemory, DefaultLinksSizeStep) { }
26
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪  indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
            ↪  memoryReservationStep, Default<LinksConstants<TLink>>.Instance,
            ↪  IndexTreeType.Default, useLinkedList: true) { }
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪  indexMemory, long memoryReservationStep, LinksConstants<TLink> constants) :
            ↪  this(dataMemory, indexMemory, memoryReservationStep, constants,
            ↪  IndexTreeType.Default, useLinkedList: true) { }
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪  indexMemory, long memoryReservationStep, LinksConstants<TLink> constants,
            ↪  IndexTreeType indexTreeType, bool useLinkedList) : base(dataMemory, indexMemory,
            ↪  memoryReservationStep, constants, useLinkedList)
35          {
36              if (indexTreeType == IndexTreeType.SizeBalancedTree)
37              {
38                  _createInternalSourceTreeMethods = () => new
                    ↪  InternalLinksSourcesSizeBalancedTreeMethods<TLink>(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
39                  _createExternalSourceTreeMethods = () => new
                    ↪  ExternalLinksSourcesSizeBalancedTreeMethods<TLink>(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
40                  _createInternalTargetTreeMethods = () => new
                    ↪  InternalLinksTargetsSizeBalancedTreeMethods<TLink>(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
41                  _createExternalTargetTreeMethods = () => new
                    ↪  ExternalLinksTargetsSizeBalancedTreeMethods<TLink>(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
42              }
43              else
44              {
45                  _createInternalSourceTreeMethods = () => new
                    ↪  InternalLinksSourcesRecursionlessSizeBalancedTreeMethods<TLink>(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
46                  _createExternalSourceTreeMethods = () => new
                    ↪  ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods<TLink>(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
47                  _createInternalTargetTreeMethods = () => new
                    ↪  InternalLinksTargetsRecursionlessSizeBalancedTreeMethods<TLink>(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
48                  _createExternalTargetTreeMethods = () => new
                    ↪  ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods<TLink>(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
49              }
50              Init(dataMemory, indexMemory);
51          }
52
53          [MethodImpl(MethodImplOptions.AggressiveInlining)]
54          protected override void SetPointers(IResizableDirectMemory dataMemory,
            ↪  IResizableDirectMemory indexMemory)
55          {
56              _linksDataParts = (byte*)dataMemory.Pointer;
57              _linksIndexParts = (byte*)indexMemory.Pointer;
58              _header = _linksIndexParts;
59              if (_useLinkedList)
60              {
61                  InternalSourcesListMethods = new
                    ↪  InternalLinksSourcesLinkedListMethods<TLink>(Constants, _linksDataParts,
                    ↪  _linksIndexParts);
62              }
63              else
64              {
65                  InternalSourcesTreeMethods = _createInternalSourceTreeMethods();
66              }
67              ExternalSourcesTreeMethods = _createExternalSourceTreeMethods();
68              InternalTargetsTreeMethods = _createInternalTargetTreeMethods();
69              ExternalTargetsTreeMethods = _createExternalTargetTreeMethods();
70              UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_linksDataParts, _header);
71          }
```

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

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

```
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using Platform.Disposables;
5   using Platform.Singletons;
6   using Platform.Converters;
7   using Platform.Numbers;
8   using Platform.Memory;
9   using Platform.Data.Exceptions;
10
11  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13  namespace Platform.Data.Doublets.Memory.Split.Generic
14  {
15      public abstract class SplitMemoryLinksBase<TLink> : DisposableBase, ILinks<TLink>
16      {
17          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
18          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
19          private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
            ↪  UncheckedConverter<TLink, long>.Default;
20          private static readonly UncheckedConverter<long, TLink> _int64ToAddressConverter =
            ↪  UncheckedConverter<long, TLink>.Default;
21
22          private static readonly TLink _zero = default;
23          private static readonly TLink _one = Arithmetic.Increment(_zero);
24
25          /// <summary>Возвращает размер одной связи в байтах.</summary>
26          /// <remarks>
27          /// Используется только во вне класса, не рекомедуется использовать внутри.
28          /// Так как во вне не обязательно будет доступен unsafe C#.
29          /// </remarks>
30          public static readonly long LinkDataPartSizeInBytes = RawLinkDataPart<TLink>.SizeInBytes;
31
32          public static readonly long LinkIndexPartSizeInBytes =
            ↪  RawLinkIndexPart<TLink>.SizeInBytes;
33
34          public static readonly long LinkHeaderSizeInBytes = LinksHeader<TLink>.SizeInBytes;
35
36          public static readonly long DefaultLinksSizeStep = 1 * 1024 * 1024;
37
38          protected readonly IResizableDirectMemory _dataMemory;
39          protected readonly IResizableDirectMemory _indexMemory;
40          protected readonly bool _useLinkedList;
41          protected readonly long _dataMemoryReservationStepInBytes;
42          protected readonly long _indexMemoryReservationStepInBytes;
43
44          protected InternalLinksSourcesLinkedListMethods<TLink> InternalSourcesListMethods;
45          protected ILinksTreeMethods<TLink> InternalSourcesTreeMethods;
46          protected ILinksTreeMethods<TLink> ExternalSourcesTreeMethods;
47          protected ILinksTreeMethods<TLink> InternalTargetsTreeMethods;
48          protected ILinksTreeMethods<TLink> ExternalTargetsTreeMethods;
49          // 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
```

```
50          protected ILinksListMethods<TLink> UnusedLinksListMethods;
51
52          /// <summary>
53          /// Возвращает общее число связей находящихся в хранилище.
54          /// </summary>
55          protected virtual TLink Total
56          {
57              [MethodImpl(MethodImplOptions.AggressiveInlining)]
58              get
59              {
60                  ref var header = ref GetHeaderReference();
61                  return Subtract(header.AllocatedLinks, header.FreeLinks);
62              }
63          }
64
65          public virtual LinksConstants<TLink> Constants
66          {
67              [MethodImpl(MethodImplOptions.AggressiveInlining)]
68              get;
69          }
70
71          [MethodImpl(MethodImplOptions.AggressiveInlining)]
72          protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants, bool
            ↪ useLinkedList)
73          {
74              _dataMemory = dataMemory;
75              _indexMemory = indexMemory;
76              _dataMemoryReservationStepInBytes = memoryReservationStep * LinkDataPartSizeInBytes;
77              _indexMemoryReservationStepInBytes = memoryReservationStep *
                ↪ LinkIndexPartSizeInBytes;
78              _useLinkedList = useLinkedList;
79              Constants = constants;
80          }
81
82          [MethodImpl(MethodImplOptions.AggressiveInlining)]
83          protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪ indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
            ↪ memoryReservationStep, Default<LinksConstants<TLink>>.Instance, useLinkedList: true)
            ↪ { }
84
85          [MethodImpl(MethodImplOptions.AggressiveInlining)]
86          protected virtual void Init(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪ indexMemory)
87          {
88              // Read allocated links from header
89              if (indexMemory.ReservedCapacity < LinkHeaderSizeInBytes)
90              {
91                  indexMemory.ReservedCapacity = LinkHeaderSizeInBytes;
92              }
93              SetPointers(dataMemory, indexMemory);
94              ref var header = ref GetHeaderReference();
95              var allocatedLinks = ConvertToInt64(header.AllocatedLinks);
96              // Adjust reserved capacity
97              var minimumDataReservedCapacity = allocatedLinks * LinkDataPartSizeInBytes;
98              if (minimumDataReservedCapacity < dataMemory.UsedCapacity)
99              {
100                 minimumDataReservedCapacity = dataMemory.UsedCapacity;
101             }
102             if (minimumDataReservedCapacity < _dataMemoryReservationStepInBytes)
103             {
104                 minimumDataReservedCapacity = _dataMemoryReservationStepInBytes;
105             }
106             var minimumIndexReservedCapacity = allocatedLinks * LinkDataPartSizeInBytes;
107             if (minimumIndexReservedCapacity < indexMemory.UsedCapacity)
108             {
109                 minimumIndexReservedCapacity = indexMemory.UsedCapacity;
110             }
111             if (minimumIndexReservedCapacity < _indexMemoryReservationStepInBytes)
112             {
113                 minimumIndexReservedCapacity = _indexMemoryReservationStepInBytes;
114             }
115             // Check for alignment
116             if (minimumDataReservedCapacity % _dataMemoryReservationStepInBytes > 0)
117             {
118                 minimumDataReservedCapacity = ((minimumDataReservedCapacity /
                    ↪ _dataMemoryReservationStepInBytes) * _dataMemoryReservationStepInBytes) +
                    ↪ _dataMemoryReservationStepInBytes;
119             }
```

```
120             if (minimumIndexReservedCapacity % _indexMemoryReservationStepInBytes > 0)
121             {
122                 minimumIndexReservedCapacity = ((minimumIndexReservedCapacity /
                    ↪  _indexMemoryReservationStepInBytes) * _indexMemoryReservationStepInBytes) +
                    ↪  _indexMemoryReservationStepInBytes;
123             }
124             if (dataMemory.ReservedCapacity != minimumDataReservedCapacity)
125             {
126                 dataMemory.ReservedCapacity = minimumDataReservedCapacity;
127             }
128             if (indexMemory.ReservedCapacity != minimumIndexReservedCapacity)
129             {
130                 indexMemory.ReservedCapacity = minimumIndexReservedCapacity;
131             }
132             SetPointers(dataMemory, indexMemory);
133             header = ref GetHeaderReference();
134             // Ensure correctness _memory.UsedCapacity over _header->AllocatedLinks
135             // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
136             dataMemory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) *
                ↪  LinkDataPartSizeInBytes) + LinkDataPartSizeInBytes; // First link is read only
                ↪  zero link.
137             indexMemory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) *
                ↪  LinkIndexPartSizeInBytes) + LinkHeaderSizeInBytes;
138             // Ensure correctness _memory.ReservedLinks over _header->ReservedCapacity
139             // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
140             header.ReservedLinks = ConvertToAddress((dataMemory.ReservedCapacity -
                ↪  LinkDataPartSizeInBytes) / LinkDataPartSizeInBytes);
141         }
142
143         [MethodImpl(MethodImplOptions.AggressiveInlining)]
144         public virtual TLink Count(IList<TLink> restrictions)
145         {
146             // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
147             if (restrictions.Count == 0)
148             {
149                 return Total;
150             }
151             var constants = Constants;
152             var any = constants.Any;
153             var index = restrictions[constants.IndexPart];
154             if (restrictions.Count == 1)
155             {
156                 if (AreEqual(index, any))
157                 {
158                     return Total;
159                 }
160                 return Exists(index) ? GetOne() : GetZero();
161             }
162             if (restrictions.Count == 2)
163             {
164                 var value = restrictions[1];
165                 if (AreEqual(index, any))
166                 {
167                     if (AreEqual(value, any))
168                     {
169                         return Total; // Any - как отсутствие ограничения
170                     }
171                     var externalReferencesRange = constants.ExternalReferencesRange;
172                     if (externalReferencesRange.HasValue &&
                        ↪  externalReferencesRange.Value.Contains(value))
173                     {
174                         return Add(ExternalSourcesTreeMethods.CountUsages(value),
                            ↪  ExternalTargetsTreeMethods.CountUsages(value));
175                     }
176                     else
177                     {
178                         if (_useLinkedList)
179                         {
180                             return Add(InternalSourcesListMethods.CountUsages(value),
                                ↪  InternalTargetsTreeMethods.CountUsages(value));
181                         }
182                         else
183                         {
184                             return Add(InternalSourcesTreeMethods.CountUsages(value),
                                ↪  InternalTargetsTreeMethods.CountUsages(value));
185                         }
186                     }
187                 }
```

```
188                 else
189                 {
190                     if (!Exists(index))
191                     {
192                         return GetZero();
193                     }
194                     if (AreEqual(value, any))
195                     {
196                         return GetOne();
197                     }
198                     ref var storedLinkValue = ref GetLinkDataPartReference(index);
199                     if (AreEqual(storedLinkValue.Source, value) ||
                        ↪ AreEqual(storedLinkValue.Target, value))
200                     {
201                         return GetOne();
202                     }
203                     return GetZero();
204                 }
205             }
206             if (restrictions.Count == 3)
207             {
208                 var externalReferencesRange = constants.ExternalReferencesRange;
209                 var source = restrictions[constants.SourcePart];
210                 var target = restrictions[constants.TargetPart];
211                 if (AreEqual(index, any))
212                 {
213                     if (AreEqual(source, any) && AreEqual(target, any))
214                     {
215                         return Total;
216                     }
217                     else if (AreEqual(source, any))
218                     {
219                         if (externalReferencesRange.HasValue &&
                            ↪ externalReferencesRange.Value.Contains(target))
220                         {
221                             return ExternalTargetsTreeMethods.CountUsages(target);
222                         }
223                         else
224                         {
225                             return InternalTargetsTreeMethods.CountUsages(target);
226                         }
227                     }
228                     else if (AreEqual(target, any))
229                     {
230                         if (externalReferencesRange.HasValue &&
                            ↪ externalReferencesRange.Value.Contains(source))
231                         {
232                             return ExternalSourcesTreeMethods.CountUsages(source);
233                         }
234                         else
235                         {
236                             if (_useLinkedList)
237                             {
238                                 return InternalSourcesListMethods.CountUsages(source);
239                             }
240                             else
241                             {
242                                 return InternalSourcesTreeMethods.CountUsages(source);
243                             }
244                         }
245                     }
246                     else //if(source != Any && target != Any)
247                     {
248                         // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
249                         TLink link;
250                         if (externalReferencesRange.HasValue)
251                         {
252                             if (externalReferencesRange.Value.Contains(source) &&
                                ↪ externalReferencesRange.Value.Contains(target))
253                             {
254                                 link = ExternalSourcesTreeMethods.Search(source, target);
255                             }
256                             else if (externalReferencesRange.Value.Contains(source))
257                             {
258                                 link = InternalTargetsTreeMethods.Search(source, target);
259                             }
260                             else if (externalReferencesRange.Value.Contains(target))
261                             {
```

```
                                if (_useLinkedList)
                                {
                                    link = ExternalSourcesTreeMethods.Search(source, target);
                                }
                                else
                                {
                                    link = InternalSourcesTreeMethods.Search(source, target);
                                }
                            }
                            else
                            {
                                if (_useLinkedList ||
                                ↪  GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
                                ↪  InternalTargetsTreeMethods.CountUsages(target)))
                                {
                                    link = InternalTargetsTreeMethods.Search(source, target);
                                }
                                else
                                {
                                    link = InternalSourcesTreeMethods.Search(source, target);
                                }
                            }
                        }
                        else
                        {
                            if (_useLinkedList ||
                            ↪  GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
                            ↪  InternalTargetsTreeMethods.CountUsages(target)))
                            {
                                link = InternalTargetsTreeMethods.Search(source, target);
                            }
                            else
                            {
                                link = InternalSourcesTreeMethods.Search(source, target);
                            }
                        }
                        return AreEqual(link, constants.Null) ? GetZero() : GetOne();
                    }
                }
                else
                {
                    if (!Exists(index))
                    {
                        return GetZero();
                    }
                    if (AreEqual(source, any) && AreEqual(target, any))
                    {
                        return GetOne();
                    }
                    ref var storedLinkValue = ref GetLinkDataPartReference(index);
                    if (!AreEqual(source, any) && !AreEqual(target, any))
                    {
                        if (AreEqual(storedLinkValue.Source, source) &&
                        ↪  AreEqual(storedLinkValue.Target, target))
                        {
                            return GetOne();
                        }
                        return GetZero();
                    }
                    var value = default(TLink);
                    if (AreEqual(source, any))
                    {
                        value = target;
                    }
                    if (AreEqual(target, any))
                    {
                        value = source;
                    }
                    if (AreEqual(storedLinkValue.Source, value) ||
                    ↪  AreEqual(storedLinkValue.Target, value))
                    {
                        return GetOne();
                    }
                    return GetZero();
                }
            }
            throw new NotSupportedException("Другие размеры и способы ограничений не
            ↪  поддерживаются.");
```

```
333         }
334
335         [MethodImpl(MethodImplOptions.AggressiveInlining)]
336         public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
337         {
338             var constants = Constants;
339             var @break = constants.Break;
340             if (restrictions.Count == 0)
341             {
342                 for (var link = GetOne(); LessOrEqualThan(link,
                    ↪ GetHeaderReference().AllocatedLinks); link = Increment(link))
343                 {
344                     if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
345                     {
346                         return @break;
347                     }
348                 }
349                 return @break;
350             }
351             var @continue = constants.Continue;
352             var any = constants.Any;
353             var index = restrictions[constants.IndexPart];
354             if (restrictions.Count == 1)
355             {
356                 if (AreEqual(index, any))
357                 {
358                     return Each(handler, Array.Empty<TLink>());
359                 }
360                 if (!Exists(index))
361                 {
362                     return @continue;
363                 }
364                 return handler(GetLinkStruct(index));
365             }
366             if (restrictions.Count == 2)
367             {
368                 var value = restrictions[1];
369                 if (AreEqual(index, any))
370                 {
371                     if (AreEqual(value, any))
372                     {
373                         return Each(handler, Array.Empty<TLink>());
374                     }
375                     if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
376                     {
377                         return @break;
378                     }
379                     return Each(handler, new Link<TLink>(index, any, value));
380                 }
381                 else
382                 {
383                     if (!Exists(index))
384                     {
385                         return @continue;
386                     }
387                     if (AreEqual(value, any))
388                     {
389                         return handler(GetLinkStruct(index));
390                     }
391                     ref var storedLinkValue = ref GetLinkDataPartReference(index);
392                     if (AreEqual(storedLinkValue.Source, value) ||
393                         AreEqual(storedLinkValue.Target, value))
394                     {
395                         return handler(GetLinkStruct(index));
396                     }
397                     return @continue;
398                 }
399             }
400             if (restrictions.Count == 3)
401             {
402                 var externalReferencesRange = constants.ExternalReferencesRange;
403                 var source = restrictions[constants.SourcePart];
404                 var target = restrictions[constants.TargetPart];
405                 if (AreEqual(index, any))
406                 {
407                     if (AreEqual(source, any) && AreEqual(target, any))
408                     {
409                         return Each(handler, Array.Empty<TLink>());
410                     }
```

```
411                         else if (AreEqual(source, any))
412                         {
413                             if (externalReferencesRange.HasValue &&
                                ↪  externalReferencesRange.Value.Contains(target))
414                             {
415                                 return ExternalTargetsTreeMethods.EachUsage(target, handler);
416                             }
417                             else
418                             {
419                                 return InternalTargetsTreeMethods.EachUsage(target, handler);
420                             }
421                         }
422                         else if (AreEqual(target, any))
423                         {
424                             if (externalReferencesRange.HasValue &&
                                ↪  externalReferencesRange.Value.Contains(source))
425                             {
426                                 return ExternalSourcesTreeMethods.EachUsage(source, handler);
427                             }
428                             else
429                             {
430                                 if (_useLinkedList)
431                                 {
432                                     return InternalSourcesListMethods.EachUsage(source, handler);
433                                 }
434                                 else
435                                 {
436                                     return InternalSourcesTreeMethods.EachUsage(source, handler);
437                                 }
438                             }
439                         }
440                         else //if(source != Any && target != Any)
441                         {
442                             TLink link;
443                             if (externalReferencesRange.HasValue)
444                             {
445                                 if (externalReferencesRange.Value.Contains(source) &&
                                    ↪  externalReferencesRange.Value.Contains(target))
446                                 {
447                                     link = ExternalSourcesTreeMethods.Search(source, target);
448                                 }
449                                 else if (externalReferencesRange.Value.Contains(source))
450                                 {
451                                     link = InternalTargetsTreeMethods.Search(source, target);
452                                 }
453                                 else if (externalReferencesRange.Value.Contains(target))
454                                 {
455                                     if (_useLinkedList)
456                                     {
457                                         link = ExternalSourcesTreeMethods.Search(source, target);
458                                     }
459                                     else
460                                     {
461                                         link = InternalSourcesTreeMethods.Search(source, target);
462                                     }
463                                 }
464                                 else
465                                 {
466                                     if (_useLinkedList ||
                                        ↪  GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
                                        ↪  InternalTargetsTreeMethods.CountUsages(target)))
467                                     {
468                                         link = InternalTargetsTreeMethods.Search(source, target);
469                                     }
470                                     else
471                                     {
472                                         link = InternalSourcesTreeMethods.Search(source, target);
473                                     }
474                                 }
475                             }
476                             else
477                             {
478                                 if (_useLinkedList ||
                                    ↪  GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
                                    ↪  InternalTargetsTreeMethods.CountUsages(target)))
479                                 {
480                                     link = InternalTargetsTreeMethods.Search(source, target);
481                                 }
```

```
                        else
                        {
                            link = InternalSourcesTreeMethods.Search(source, target);
                        }
                    }
                    return AreEqual(link, constants.Null) ? @continue :
                    ↪  handler(GetLinkStruct(link));
                }
            }
            else
            {
                if (!Exists(index))
                {
                    return @continue;
                }
                if (AreEqual(source, any) && AreEqual(target, any))
                {
                    return handler(GetLinkStruct(index));
                }
                ref var storedLinkValue = ref GetLinkDataPartReference(index);
                if (!AreEqual(source, any) && !AreEqual(target, any))
                {
                    if (AreEqual(storedLinkValue.Source, source) &&
                        AreEqual(storedLinkValue.Target, target))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    return @continue;
                }
                var value = default(TLink);
                if (AreEqual(source, any))
                {
                    value = target;
                }
                if (AreEqual(target, any))
                {
                    value = source;
                }
                if (AreEqual(storedLinkValue.Source, value) ||
                    AreEqual(storedLinkValue.Target, value))
                {
                    return handler(GetLinkStruct(index));
                }
                return @continue;
            }
        }
        throw new NotSupportedException("Другие размеры и способы ограничений не
        ↪  поддерживаются.");
    }

    /// <remarks>
    /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
    ↪  в другом месте (но не в менеджере памяти, а в логике Links)
    /// </remarks>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
    {
        var constants = Constants;
        var @null = constants.Null;
        var externalReferencesRange = constants.ExternalReferencesRange;
        var linkIndex = restrictions[constants.IndexPart];
        ref var link = ref GetLinkDataPartReference(linkIndex);
        var source = link.Source;
        var target = link.Target;
        ref var header = ref GetHeaderReference();
        ref var rootAsSource = ref header.RootAsSource;
        ref var rootAsTarget = ref header.RootAsTarget;
        // Будет корректно работать только в том случае, если пространство выделенной связи
        ↪  предварительно заполнено нулями
        if (!AreEqual(source, @null))
        {
            if (externalReferencesRange.HasValue &&
            ↪  externalReferencesRange.Value.Contains(source))
            {
                ExternalSourcesTreeMethods.Detach(ref rootAsSource, linkIndex);
            }
            else
            {
                if (_useLinkedList)
```

```
556                     {
557                         InternalSourcesListMethods.Detach(source, linkIndex);
558                     }
559                     else
560                     {
561                         InternalSourcesTreeMethods.Detach(ref
                            ↪ GetLinkIndexPartReference(source).RootAsSource, linkIndex);
562                     }
563                 }
564             }
565             if (!AreEqual(target, @null))
566             {
567                 if (externalReferencesRange.HasValue &&
                    ↪ externalReferencesRange.Value.Contains(target))
568                 {
569                     ExternalTargetsTreeMethods.Detach(ref rootAsTarget, linkIndex);
570                 }
571                 else
572                 {
573                     InternalTargetsTreeMethods.Detach(ref
                        ↪ GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
574                 }
575             }
576             source = link.Source = substitution[constants.SourcePart];
577             target = link.Target = substitution[constants.TargetPart];
578             if (!AreEqual(source, @null))
579             {
580                 if (externalReferencesRange.HasValue &&
                    ↪ externalReferencesRange.Value.Contains(source))
581                 {
582                     ExternalSourcesTreeMethods.Attach(ref rootAsSource, linkIndex);
583                 }
584                 else
585                 {
586                     if (_useLinkedList)
587                     {
588                         InternalSourcesListMethods.AttachAsLast(source, linkIndex);
589                     }
590                     else
591                     {
592                         InternalSourcesTreeMethods.Attach(ref
                            ↪ GetLinkIndexPartReference(source).RootAsSource, linkIndex);
593                     }
594                 }
595             }
596             if (!AreEqual(target, @null))
597             {
598                 if (externalReferencesRange.HasValue &&
                    ↪ externalReferencesRange.Value.Contains(target))
599                 {
600                     ExternalTargetsTreeMethods.Attach(ref rootAsTarget, linkIndex);
601                 }
602                 else
603                 {
604                     InternalTargetsTreeMethods.Attach(ref
                        ↪ GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
605                 }
606             }
607             return linkIndex;
608         }
609
610         /// <remarks>
611         /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
            ↪ пространство
612         /// </remarks>
613         [MethodImpl(MethodImplOptions.AggressiveInlining)]
614         public virtual TLink Create(IList<TLink> restrictions)
615         {
616             ref var header = ref GetHeaderReference();
617             var freeLink = header.FirstFreeLink;
618             if (!AreEqual(freeLink, Constants.Null))
619             {
620                 UnusedLinksListMethods.Detach(freeLink);
621             }
622             else
623             {
624                 var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
625                 if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
```

```
626                 {
627                     throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
628                 }
629                 if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
630                 {
631                     _dataMemory.ReservedCapacity += _dataMemoryReservationStepInBytes;
632                     _indexMemory.ReservedCapacity += _indexMemoryReservationStepInBytes;
633                     SetPointers(_dataMemory, _indexMemory);
634                     header = ref GetHeaderReference();
635                     header.ReservedLinks = ConvertToAddress(_dataMemory.ReservedCapacity /
                        ↪  LinkDataPartSizeInBytes);
636                 }
637                 freeLink = header.AllocatedLinks = Increment(header.AllocatedLinks);
638                 _dataMemory.UsedCapacity += LinkDataPartSizeInBytes;
639                 _indexMemory.UsedCapacity += LinkIndexPartSizeInBytes;
640             }
641             return freeLink;
642         }
643
644         [MethodImpl(MethodImplOptions.AggressiveInlining)]
645         public virtual void Delete(IList<TLink> restrictions)
646         {
647             ref var header = ref GetHeaderReference();
648             var link = restrictions[Constants.IndexPart];
649             if (LessThan(link, header.AllocatedLinks))
650             {
651                 UnusedLinksListMethods.AttachAsFirst(link);
652             }
653             else if (AreEqual(link, header.AllocatedLinks))
654             {
655                 header.AllocatedLinks = Decrement(header.AllocatedLinks);
656                 _dataMemory.UsedCapacity -= LinkDataPartSizeInBytes;
657                 _indexMemory.UsedCapacity -= LinkIndexPartSizeInBytes;
658                 // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
                    ↪  пока не дойдём до первой существующей связи
659                 // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
660                 while (GreaterThan(header.AllocatedLinks, GetZero()) &&
                    ↪  IsUnusedLink(header.AllocatedLinks))
661                 {
662                     UnusedLinksListMethods.Detach(header.AllocatedLinks);
663                     header.AllocatedLinks = Decrement(header.AllocatedLinks);
664                     _dataMemory.UsedCapacity -= LinkDataPartSizeInBytes;
665                     _indexMemory.UsedCapacity -= LinkIndexPartSizeInBytes;
666                 }
667             }
668         }
669
670         [MethodImpl(MethodImplOptions.AggressiveInlining)]
671         public IList<TLink> GetLinkStruct(TLink linkIndex)
672         {
673             ref var link = ref GetLinkDataPartReference(linkIndex);
674             return new Link<TLink>(linkIndex, link.Source, link.Target);
675         }
676
677         /// <remarks>
678         /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
                ↪  адрес реально поменялся
679         ///
680         /// Указатель this.links может быть в том же месте,
681         /// так как 0-я связь не используется и имеет такой же размер как Header,
682         /// поэтому header размещается в том же месте, что и 0-я связь
683         /// </remarks>
684         [MethodImpl(MethodImplOptions.AggressiveInlining)]
685         protected abstract void SetPointers(IResizableDirectMemory dataMemory,
                ↪  IResizableDirectMemory indexMemory);
686
687         [MethodImpl(MethodImplOptions.AggressiveInlining)]
688         protected virtual void ResetPointers()
689         {
690             InternalSourcesListMethods = null;
691             InternalSourcesTreeMethods = null;
692             ExternalSourcesTreeMethods = null;
693             InternalTargetsTreeMethods = null;
694             ExternalTargetsTreeMethods = null;
695             UnusedLinksListMethods = null;
696         }
697
698         [MethodImpl(MethodImplOptions.AggressiveInlining)]
699         protected abstract ref LinksHeader<TLink> GetHeaderReference();
```

```
700
701         [MethodImpl(MethodImplOptions.AggressiveInlining)]
702         protected abstract ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex);
703
704         [MethodImpl(MethodImplOptions.AggressiveInlining)]
705         protected abstract ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
            ↪ linkIndex);
706
707         [MethodImpl(MethodImplOptions.AggressiveInlining)]
708         protected virtual bool Exists(TLink link)
709             => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
710             && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
711             && !IsUnusedLink(link);
712
713         [MethodImpl(MethodImplOptions.AggressiveInlining)]
714         protected virtual bool IsUnusedLink(TLink linkIndex)
715         {
716             if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
                ↪ is not needed
717             {
718                 // TODO: Reduce access to memory in different location (should be enough to use
                    ↪ just linkIndexPart)
719                 ref var linkDataPart = ref GetLinkDataPartReference(linkIndex);
720                 ref var linkIndexPart = ref GetLinkIndexPartReference(linkIndex);
721                 return AreEqual(linkIndexPart.SizeAsTarget, default) &&
                    ↪ !AreEqual(linkDataPart.Source, default);
722             }
723             else
724             {
725                 return true;
726             }
727         }
728
729         [MethodImpl(MethodImplOptions.AggressiveInlining)]
730         protected virtual TLink GetOne() => _one;
731
732         [MethodImpl(MethodImplOptions.AggressiveInlining)]
733         protected virtual TLink GetZero() => default;
734
735         [MethodImpl(MethodImplOptions.AggressiveInlining)]
736         protected virtual bool AreEqual(TLink first, TLink second) =>
            ↪ _equalityComparer.Equals(first, second);
737
738         [MethodImpl(MethodImplOptions.AggressiveInlining)]
739         protected virtual bool LessThan(TLink first, TLink second) => _comparer.Compare(first,
            ↪ second) < 0;
740
741         [MethodImpl(MethodImplOptions.AggressiveInlining)]
742         protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
            ↪ _comparer.Compare(first, second) <= 0;
743
744         [MethodImpl(MethodImplOptions.AggressiveInlining)]
745         protected virtual bool GreaterThan(TLink first, TLink second) =>
            ↪ _comparer.Compare(first, second) > 0;
746
747         [MethodImpl(MethodImplOptions.AggressiveInlining)]
748         protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
            ↪ _comparer.Compare(first, second) >= 0;
749
750         [MethodImpl(MethodImplOptions.AggressiveInlining)]
751         protected virtual long ConvertToInt64(TLink value) =>
            ↪ _addressToInt64Converter.Convert(value);
752
753         [MethodImpl(MethodImplOptions.AggressiveInlining)]
754         protected virtual TLink ConvertToAddress(long value) =>
            ↪ _int64ToAddressConverter.Convert(value);
755
756         [MethodImpl(MethodImplOptions.AggressiveInlining)]
757         protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
            ↪ second);
758
759         [MethodImpl(MethodImplOptions.AggressiveInlining)]
760         protected virtual TLink Subtract(TLink first, TLink second) =>
            ↪ Arithmetic<TLink>.Subtract(first, second);
761
762         [MethodImpl(MethodImplOptions.AggressiveInlining)]
763         protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);
764
```

```
765         [MethodImpl(MethodImplOptions.AggressiveInlining)]
766         protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);
767 
768         #region Disposable
769 
770         protected override bool AllowMultipleDisposeCalls
771         {
772             [MethodImpl(MethodImplOptions.AggressiveInlining)]
773             get => true;
774         }
775 
776         [MethodImpl(MethodImplOptions.AggressiveInlining)]
777         protected override void Dispose(bool manual, bool wasDisposed)
778         {
779             if (!wasDisposed)
780             {
781                 ResetPointers();
782                 _dataMemory.DisposeIfPossible();
783                 _indexMemory.DisposeIfPossible();
784             }
785         }
786 
787         #endregion
788     }
789 }
```

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

```
1  using System.Runtime.CompilerServices;
2  using Platform.Collections.Methods.Lists;
3  using Platform.Converters;
4  using static System.Runtime.CompilerServices.Unsafe;
5 
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7 
8  namespace Platform.Data.Doublets.Memory.Split.Generic
9  {
10     public unsafe class UnusedLinksListMethods<TLink> :
       ↪  AbsoluteCircularDoublyLinkedListMethods<TLink>, ILinksListMethods<TLink>
11     {
12         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
           ↪  UncheckedConverter<TLink, long>.Default;
13 
14         private readonly byte* _links;
15         private readonly byte* _header;
16 
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public UnusedLinksListMethods(byte* links, byte* header)
19         {
20             _links = links;
21             _header = header;
22         }
23 
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
           ↪  AsRef<LinksHeader<TLink>>(_header);
26 
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
           ↪  AsRef<RawLinkDataPart<TLink>>(_links + (RawLinkDataPart<TLink>.SizeInBytes *
           ↪  _addressToInt64Converter.Convert(link)));
29 
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected override TLink GetFirst() => GetHeaderReference().FirstFreeLink;
32 
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected override TLink GetLast() => GetHeaderReference().LastFreeLink;
35 
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override TLink GetPrevious(TLink element) =>
           ↪  GetLinkDataPartReference(element).Source;
38 
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override TLink GetNext(TLink element) =>
           ↪  GetLinkDataPartReference(element).Target;
41 
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected override TLink GetSize() => GetHeaderReference().FreeLinks;
44 
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
46          protected override void SetFirst(TLink element) => GetHeaderReference().FirstFreeLink =
            ↪  element;
47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          protected override void SetLast(TLink element) => GetHeaderReference().LastFreeLink =
            ↪  element;
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override void SetPrevious(TLink element, TLink previous) =>
            ↪  GetLinkDataPartReference(element).Source = previous;
53
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          protected override void SetNext(TLink element, TLink next) =>
            ↪  GetLinkDataPartReference(element).Target = next;
56
57          [MethodImpl(MethodImplOptions.AggressiveInlining)]
58          protected override void SetSize(TLink size) => GetHeaderReference().FreeLinks = size;
59      }
60  }
```

1.45 ./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      public struct RawLinkDataPart<TLink> : IEquatable<RawLinkDataPart<TLink>>
11      {
12          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
13
14          public static readonly long SizeInBytes = Structure<RawLinkDataPart<TLink>>.Size;
15
16          public TLink Source;
17          public TLink Target;
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          public override bool Equals(object obj) => obj is RawLinkDataPart<TLink> link ?
            ↪  Equals(link) : false;
21
22          [MethodImpl(MethodImplOptions.AggressiveInlining)]
23          public bool Equals(RawLinkDataPart<TLink> other)
24              => _equalityComparer.Equals(Source, other.Source)
25              && _equalityComparer.Equals(Target, other.Target);
26
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          public override int GetHashCode() => (Source, Target).GetHashCode();
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public static bool operator ==(RawLinkDataPart<TLink> left, RawLinkDataPart<TLink>
            ↪  right) => left.Equals(right);
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          public static bool operator !=(RawLinkDataPart<TLink> left, RawLinkDataPart<TLink>
            ↪  right) => !(left == right);
35      }
36  }
```

1.46 ./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      public struct RawLinkIndexPart<TLink> : IEquatable<RawLinkIndexPart<TLink>>
11      {
12          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
13
14          public static readonly long SizeInBytes = Structure<RawLinkIndexPart<TLink>>.Size;
15
16          public TLink RootAsSource;
```

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

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

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

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

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

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

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

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

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe abstract class UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase :
    ↪  ExternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>, ILinksTreeMethods<TLink>
    {
        protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
        protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
        protected new readonly LinksHeader<TLink>* Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected
        ↪  UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLink>
        ↪  constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
        ↪  linksIndexParts, LinksHeader<TLink>* header)
            : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
        {
            LinksDataParts = linksDataParts;
            LinksIndexParts = linksIndexParts;
            Header = header;
        }

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

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

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

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

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

```

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

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

```
1   using System.Runtime.CompilerServices;
2   using Platform.Data.Doublets.Memory.Split.Generic;
3   using TLink = System.UInt32;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.Memory.Split.Specific
8   {
9       public unsafe abstract class UInt32ExternalLinksSizeBalancedTreeMethodsBase :
        ↪  ExternalLinksSizeBalancedTreeMethodsBase<TLink>, ILinksTreeMethods<TLink>
10      {
11          protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
12          protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
13          protected new readonly LinksHeader<TLink>* Header;
```

```
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          protected UInt32ExternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLink>
            ↪  constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
            ↪  linksIndexParts, LinksHeader<TLink>* header)
17              : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
18          {
19              LinksDataParts = linksDataParts;
20              LinksIndexParts = linksIndexParts;
21              Header = header;
22          }
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          protected override TLink GetZero() => 0U;
26
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          protected override bool EqualToZero(TLink value) => value == 0U;
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          protected override bool AreEqual(TLink first, TLink second) => first == second;
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          protected override bool GreaterThanZero(TLink value) => value > 0U;
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          protected override bool GreaterThan(TLink first, TLink second) => first > second;
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          protected override bool GreaterOrEqualThan(TLink first, TLink second) => first >= second;
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          protected override bool GreaterOrEqualThanZero(TLink value) => true; // value >= 0 is
            ↪  always true for ulong
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected override bool LessOrEqualThanZero(TLink value) => value == 0UL; // value is
            ↪  always >= 0 for ulong
47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          protected override bool LessOrEqualThan(TLink first, TLink second) => first <= second;
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override bool LessThanZero(TLink value) => false; // value < 0 is always false
            ↪  for ulong
53
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          protected override bool LessThan(TLink first, TLink second) => first < second;
56
57          [MethodImpl(MethodImplOptions.AggressiveInlining)]
58          protected override TLink Increment(TLink value) => ++value;
59
60          [MethodImpl(MethodImplOptions.AggressiveInlining)]
61          protected override TLink Decrement(TLink value) => --value;
62
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          protected override TLink Add(TLink first, TLink second) => first + second;
65
66          [MethodImpl(MethodImplOptions.AggressiveInlining)]
67          protected override TLink Subtract(TLink first, TLink second) => first - second;
68
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          protected override ref LinksHeader<TLink> GetHeaderReference() => ref *Header;
71
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
            ↪  ref LinksDataParts[link];
74
75          [MethodImpl(MethodImplOptions.AggressiveInlining)]
76          protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
            ↪  ref LinksIndexParts[link];
77
78          [MethodImpl(MethodImplOptions.AggressiveInlining)]
79          protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
80          {
81              ref var firstLink = ref LinksDataParts[first];
82              ref var secondLink = ref LinksDataParts[second];
83              return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
84          }
```

```
85
86          [MethodImpl(MethodImplOptions.AggressiveInlining)]
87          protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
88          {
89              ref var firstLink = ref LinksDataParts[first];
90              ref var secondLink = ref LinksDataParts[second];
91              return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
92          }
93      }
94  }
```

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

```
1   using System.Runtime.CompilerServices;
2   using TLink = System.UInt32;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Memory.Split.Specific
7   {
8       public unsafe class UInt32ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods :
        ↪  UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase
9       {
10          [MethodImpl(MethodImplOptions.AggressiveInlining)]
11          public
            ↪  UInt32ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
            ↪  constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
            ↪  linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
            ↪  linksIndexParts, header) { }
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          protected override ref TLink GetLeftReference(TLink node) => ref
            ↪  LinksIndexParts[node].LeftAsSource;
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          protected override ref TLink GetRightReference(TLink node) => ref
            ↪  LinksIndexParts[node].RightAsSource;
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsSource;
21
22          [MethodImpl(MethodImplOptions.AggressiveInlining)]
23          protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsSource;
24
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          protected override void SetLeft(TLink node, TLink left) =>
            ↪  LinksIndexParts[node].LeftAsSource = left;
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          protected override void SetRight(TLink node, TLink right) =>
            ↪  LinksIndexParts[node].RightAsSource = right;
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsSource;
33
34          [MethodImpl(MethodImplOptions.AggressiveInlining)]
35          protected override void SetSize(TLink node, TLink size) =>
            ↪  LinksIndexParts[node].SizeAsSource = size;
36
37          [MethodImpl(MethodImplOptions.AggressiveInlining)]
38          protected override TLink GetTreeRoot() => Header->RootAsSource;
39
40          [MethodImpl(MethodImplOptions.AggressiveInlining)]
41          protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Source;
42
43          [MethodImpl(MethodImplOptions.AggressiveInlining)]
44          protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
            ↪  TLink secondSource, TLink secondTarget)
                => firstSource < secondSource || firstSource == secondSource && firstTarget <
                ↪  secondTarget;
46
47          [MethodImpl(MethodImplOptions.AggressiveInlining)]
48          protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
            ↪  TLink secondSource, TLink secondTarget)
                => firstSource > secondSource || firstSource == secondSource && firstTarget >
                ↪  secondTarget;
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override void ClearNode(TLink node)
```

```
53          {
54              ref var link = ref LinksIndexParts[node];
55              link.LeftAsSource = Zero;
56              link.RightAsSource = Zero;
57              link.SizeAsSource = Zero;
58          }
59      }
60  }
```

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

```
1   using System.Runtime.CompilerServices;
2   using TLink = System.UInt32;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Memory.Split.Specific
7   {
8       public unsafe class UInt32ExternalLinksSourcesSizeBalancedTreeMethods :
        ↪  UInt32ExternalLinksSizeBalancedTreeMethodsBase
9       {
10          [MethodImpl(MethodImplOptions.AggressiveInlining)]
11          public UInt32ExternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink>
            ↪  constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
            ↪  linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
            ↪  linksIndexParts, header) { }
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          protected override ref TLink GetLeftReference(TLink node) => ref
            ↪  LinksIndexParts[node].LeftAsSource;
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          protected override ref TLink GetRightReference(TLink node) => ref
            ↪  LinksIndexParts[node].RightAsSource;
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsSource;
21
22          [MethodImpl(MethodImplOptions.AggressiveInlining)]
23          protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsSource;
24
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          protected override void SetLeft(TLink node, TLink left) =>
            ↪  LinksIndexParts[node].LeftAsSource = left;
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          protected override void SetRight(TLink node, TLink right) =>
            ↪  LinksIndexParts[node].RightAsSource = right;
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsSource;
33
34          [MethodImpl(MethodImplOptions.AggressiveInlining)]
35          protected override void SetSize(TLink node, TLink size) =>
            ↪  LinksIndexParts[node].SizeAsSource = size;
36
37          [MethodImpl(MethodImplOptions.AggressiveInlining)]
38          protected override TLink GetTreeRoot() => Header->RootAsSource;
39
40          [MethodImpl(MethodImplOptions.AggressiveInlining)]
41          protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Source;
42
43          [MethodImpl(MethodImplOptions.AggressiveInlining)]
44          protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
            ↪  TLink secondSource, TLink secondTarget)
45              => firstSource < secondSource || firstSource == secondSource && firstTarget <
                ↪  secondTarget;
46
47          [MethodImpl(MethodImplOptions.AggressiveInlining)]
48          protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
            ↪  TLink secondSource, TLink secondTarget)
49              => firstSource > secondSource || firstSource == secondSource && firstTarget >
                ↪  secondTarget;
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override void ClearNode(TLink node)
53          {
54              ref var link = ref LinksIndexParts[node];
55              link.LeftAsSource = Zero;
56              link.RightAsSource = Zero;
```

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

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

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

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe class UInt32ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods :
    ↪ UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public
        ↪ UInt32ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
        ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
        ↪ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
        ↪ linksIndexParts, header) { }

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

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

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

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

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

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

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

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

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

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

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

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

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

```
}
```

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

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

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe class UInt32ExternalLinksTargetsSizeBalancedTreeMethods :
    ↪ UInt32ExternalLinksSizeBalancedTreeMethodsBase
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32ExternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink>
        ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
        ↪ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
        ↪ linksIndexParts, header) { }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe abstract class UInt32InternalLinksRecursionlessSizeBalancedTreeMethodsBase :
    ↪ InternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
    {
        protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
        protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
        protected new readonly LinksHeader<TLink>* Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected
        ↪ UInt32InternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLink>
        ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
        ↪ linksIndexParts, LinksHeader<TLink>* header)
            : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
        {
            LinksDataParts = linksDataParts;
            LinksIndexParts = linksIndexParts;
            Header = header;
        }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe abstract class UInt32InternalLinksSizeBalancedTreeMethodsBase :
    ↪ InternalLinksSizeBalancedTreeMethodsBase<TLink>
    {
        protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
        protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
        protected new readonly LinksHeader<TLink>* Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected UInt32InternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLink>
        ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
        ↪ linksIndexParts, LinksHeader<TLink>* header)
            : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
        {
            LinksDataParts = linksDataParts;
            LinksIndexParts = linksIndexParts;
            Header = header;
        }

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

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

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

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

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

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

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

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

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

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

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

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

```

```
60          [MethodImpl(MethodImplOptions.AggressiveInlining)]
61          protected override TLink Decrement(TLink value) => --value;
62
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          protected override TLink Add(TLink first, TLink second) => first + second;
65
66          [MethodImpl(MethodImplOptions.AggressiveInlining)]
67          protected override TLink Subtract(TLink first, TLink second) => first - second;
68
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
            ↪  ref LinksDataParts[link];
71
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
            ↪  ref LinksIndexParts[link];
74
75          [MethodImpl(MethodImplOptions.AggressiveInlining)]
76          protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second) =>
            ↪  GetKeyPartValue(first) < GetKeyPartValue(second);
77
78          [MethodImpl(MethodImplOptions.AggressiveInlining)]
79          protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second) =>
            ↪  GetKeyPartValue(first) > GetKeyPartValue(second);
80      }
81  }
```

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

```
1   using System.Runtime.CompilerServices;
2   using TLink = System.UInt32;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Memory.Split.Generic
7   {
8       public unsafe class UInt32InternalLinksSourcesLinkedListMethods :
            ↪  InternalLinksSourcesLinkedListMethods<TLink>
9       {
10          private readonly RawLinkDataPart<TLink>* _linksDataParts;
11          private readonly RawLinkIndexPart<TLink>* _linksIndexParts;
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          public UInt32InternalLinksSourcesLinkedListMethods(LinksConstants<TLink> constants,
                ↪  RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>* linksIndexParts)
15              : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts)
16          {
17              _linksDataParts = linksDataParts;
18              _linksIndexParts = linksIndexParts;
19          }
20
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
                ↪  ref _linksDataParts[link];
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
                ↪  ref _linksIndexParts[link];
26      }
27  }
```

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

```
1   using System.Runtime.CompilerServices;
2   using TLink = System.UInt32;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Memory.Split.Specific
7   {
8       public unsafe class UInt32InternalLinksSourcesRecursionlessSizeBalancedTreeMethods :
            ↪  UInt32InternalLinksRecursionlessSizeBalancedTreeMethodsBase
9       {
10          [MethodImpl(MethodImplOptions.AggressiveInlining)]
11          public
                ↪  UInt32InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
                ↪  constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
                ↪  linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
                ↪  linksIndexParts, header) { }
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe class UInt32InternalLinksTargetsSizeBalancedTreeMethods :
    ↪  UInt32InternalLinksSizeBalancedTreeMethodsBase
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32InternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink>
        ↪  constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
        ↪  linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
        ↪  linksIndexParts, header) { }

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe class UInt32SplitMemoryLinks : SplitMemoryLinksBase<TLink>
    {
        private readonly Func<ILinksTreeMethods<TLink>> _createInternalSourceTreeMethods;
        private readonly Func<ILinksTreeMethods<TLink>> _createExternalSourceTreeMethods;
        private readonly Func<ILinksTreeMethods<TLink>> _createInternalTargetTreeMethods;
        private readonly Func<ILinksTreeMethods<TLink>> _createExternalTargetTreeMethods;
        private LinksHeader<TLink>* _header;
        private RawLinkDataPart<TLink>* _linksDataParts;
        private RawLinkIndexPart<TLink>* _linksIndexParts;

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt32SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
        ↪ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants,
        ↪ IndexTreeType indexTreeType, bool useLinkedList) : base(dataMemory, indexMemory,
        ↪ memoryReservationStep, constants, useLinkedList)
        {
            if (indexTreeType == IndexTreeType.SizeBalancedTree)
            {
                _createInternalSourceTreeMethods = () => new
                ↪ UInt32InternalLinksSourcesSizeBalancedTreeMethods(Constants,
                ↪ _linksDataParts, _linksIndexParts, _header);
                _createExternalSourceTreeMethods = () => new
                ↪ UInt32ExternalLinksSourcesSizeBalancedTreeMethods(Constants,
                ↪ _linksDataParts, _linksIndexParts, _header);
                _createInternalTargetTreeMethods = () => new
                ↪ UInt32InternalLinksTargetsSizeBalancedTreeMethods(Constants,
                ↪ _linksDataParts, _linksIndexParts, _header);
                _createExternalTargetTreeMethods = () => new
                ↪ UInt32ExternalLinksTargetsSizeBalancedTreeMethods(Constants,
                ↪ _linksDataParts, _linksIndexParts, _header);
```

```
40              }
41              else
42              {
43                  _createInternalSourceTreeMethods = () => new
                    ↪  UInt32InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
44                  _createExternalSourceTreeMethods = () => new
                    ↪  UInt32ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
45                  _createInternalTargetTreeMethods = () => new
                    ↪  UInt32InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
46                  _createExternalTargetTreeMethods = () => new
                    ↪  UInt32ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
47              }
48              Init(dataMemory, indexMemory);
49          }
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override void SetPointers(IResizableDirectMemory dataMemory,
            ↪  IResizableDirectMemory indexMemory)
53          {
54              _linksDataParts = (RawLinkDataPart<TLink>*)dataMemory.Pointer;
55              _linksIndexParts = (RawLinkIndexPart<TLink>*)indexMemory.Pointer;
56              _header = (LinksHeader<TLink>*)indexMemory.Pointer;
57              if (_useLinkedList)
58              {
59                  InternalSourcesListMethods = new
                    ↪  UInt32InternalLinksSourcesLinkedListMethods(Constants, _linksDataParts,
                    ↪  _linksIndexParts);
60              }
61              else
62              {
63                  InternalSourcesTreeMethods = _createInternalSourceTreeMethods();
64              }
65              ExternalSourcesTreeMethods = _createExternalSourceTreeMethods();
66              InternalTargetsTreeMethods = _createInternalTargetTreeMethods();
67              ExternalTargetsTreeMethods = _createExternalTargetTreeMethods();
68              UnusedLinksListMethods = new UInt32UnusedLinksListMethods(_linksDataParts, _header);
69          }
70
71          [MethodImpl(MethodImplOptions.AggressiveInlining)]
72          protected override void ResetPointers()
73          {
74              base.ResetPointers();
75              _linksDataParts = null;
76              _linksIndexParts = null;
77              _header = null;
78          }
79
80          [MethodImpl(MethodImplOptions.AggressiveInlining)]
81          protected override ref LinksHeader<TLink> GetHeaderReference() => ref *_header;
82
83          [MethodImpl(MethodImplOptions.AggressiveInlining)]
84          protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex)
            ↪  => ref _linksDataParts[linkIndex];
85
86          [MethodImpl(MethodImplOptions.AggressiveInlining)]
87          protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
            ↪  linkIndex) => ref _linksIndexParts[linkIndex];
88
89          [MethodImpl(MethodImplOptions.AggressiveInlining)]
90          protected override bool AreEqual(TLink first, TLink second) => first == second;
91
92          [MethodImpl(MethodImplOptions.AggressiveInlining)]
93          protected override bool LessThan(TLink first, TLink second) => first < second;
94
95          [MethodImpl(MethodImplOptions.AggressiveInlining)]
96          protected override bool LessOrEqualThan(TLink first, TLink second) => first <= second;
97
98          [MethodImpl(MethodImplOptions.AggressiveInlining)]
99          protected override bool GreaterThan(TLink first, TLink second) => first > second;
100
101         [MethodImpl(MethodImplOptions.AggressiveInlining)]
102         protected override bool GreaterOrEqualThan(TLink first, TLink second) => first >= second;
103
104         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        protected override TLink GetZero() => 0U;

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

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

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

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

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

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

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

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

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

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe class UInt32UnusedLinksListMethods : UnusedLinksListMethods<TLink>
    {
        private readonly RawLinkDataPart<TLink>* _links;
        private readonly LinksHeader<TLink>* _header;

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

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

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

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

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

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe abstract class UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase :
    ↪  ExternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>, ILinksTreeMethods<TLink>
    {
        protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
        protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
        protected new readonly LinksHeader<TLink>* Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected
        ↪  UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLink>
        ↪  constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
        ↪  linksIndexParts, LinksHeader<TLink>* header)
            : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
        {
```

```
            LinksDataParts = linksDataParts;
            LinksIndexParts = linksIndexParts;
            Header = header;
        }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
            return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
            ↪  secondLink.Source, secondLink.Target);
        }
    }
}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

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

```
1   using System.Runtime.CompilerServices;
2   using TLink = System.UInt64;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Memory.Split.Specific
7   {
8       public unsafe class UInt64ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods :
        ↪  UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase
9       {
10          [MethodImpl(MethodImplOptions.AggressiveInlining)]
11          public
            ↪  UInt64ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
            ↪  constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
            ↪  linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
            ↪  linksIndexParts, header) { }
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          protected override ref TLink GetLeftReference(TLink node) => ref
            ↪  LinksIndexParts[node].LeftAsTarget;
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          protected override ref TLink GetRightReference(TLink node) => ref
            ↪  LinksIndexParts[node].RightAsTarget;
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsTarget;
21
22          [MethodImpl(MethodImplOptions.AggressiveInlining)]
23          protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsTarget;
24
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          protected override void SetLeft(TLink node, TLink left) =>
            ↪  LinksIndexParts[node].LeftAsTarget = left;
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          protected override void SetRight(TLink node, TLink right) =>
            ↪  LinksIndexParts[node].RightAsTarget = right;
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsTarget;
33
34          [MethodImpl(MethodImplOptions.AggressiveInlining)]
35          protected override void SetSize(TLink node, TLink size) =>
            ↪  LinksIndexParts[node].SizeAsTarget = size;
36
37          [MethodImpl(MethodImplOptions.AggressiveInlining)]
38          protected override TLink GetTreeRoot() => Header->RootAsTarget;
39
40          [MethodImpl(MethodImplOptions.AggressiveInlining)]
41          protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Target;
42
43          [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
        ↪  TLink secondSource, TLink secondTarget)
            => firstTarget < secondTarget || firstTarget == secondTarget && firstSource <
            ↪  secondSource;

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

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

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

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

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe class UInt64ExternalLinksTargetsSizeBalancedTreeMethods :
    ↪  UInt64ExternalLinksSizeBalancedTreeMethodsBase
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64ExternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink>
        ↪  constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
        ↪  linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
        ↪  linksIndexParts, header) { }

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

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

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

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

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

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

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

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

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

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

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

```
46
47          [MethodImpl(MethodImplOptions.AggressiveInlining)]
48          protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
            ↪  TLink secondSource, TLink secondTarget)
49              => firstTarget > secondTarget || firstTarget == secondTarget && firstSource >
                ↪  secondSource;
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override void ClearNode(TLink node)
53          {
54              ref var link = ref LinksIndexParts[node];
55              link.LeftAsTarget = Zero;
56              link.RightAsTarget = Zero;
57              link.SizeAsTarget = Zero;
58          }
59      }
60  }
```

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

```
1   using System.Runtime.CompilerServices;
2   using Platform.Data.Doublets.Memory.Split.Generic;
3   using TLink = System.UInt64;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.Memory.Split.Specific
8   {
9       public unsafe abstract class UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase :
        ↪  InternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
10      {
11          protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
12          protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
13          protected new readonly LinksHeader<TLink>* Header;
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          protected
            ↪  UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLink>
            ↪  constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
            ↪  linksIndexParts, LinksHeader<TLink>* header)
17              : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
18          {
19              LinksDataParts = linksDataParts;
20              LinksIndexParts = linksIndexParts;
21              Header = header;
22          }
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          protected override ulong GetZero() => 0UL;
26
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          protected override bool EqualToZero(ulong value) => value == 0UL;
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          protected override bool AreEqual(ulong first, ulong second) => first == second;
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          protected override bool GreaterThanZero(ulong value) => value > 0UL;
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          protected override bool GreaterThan(ulong first, ulong second) => first > second;
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
            ↪  always true for ulong
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
            ↪  always >= 0 for ulong
47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
            ↪  for ulong
53
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

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

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

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

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

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

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

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

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

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

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

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe abstract class UInt64InternalLinksSizeBalancedTreeMethodsBase :
    ↪  InternalLinksSizeBalancedTreeMethodsBase<TLink>
    {
        protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
        protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
        protected new readonly LinksHeader<TLink>* Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected UInt64InternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLink>
        ↪  constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
        ↪  linksIndexParts, LinksHeader<TLink>* header)
            : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
        {
            LinksDataParts = linksDataParts;
            LinksIndexParts = linksIndexParts;
            Header = header;
        }

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe class UInt64InternalLinksSourcesRecursionlessSizeBalancedTreeMethods :
    ↪ UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public
        ↪ UInt64InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
        ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
        ↪ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
        ↪ linksIndexParts, header) { }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe class UInt64InternalLinksSourcesSizeBalancedTreeMethods :
    ↪ UInt64InternalLinksSizeBalancedTreeMethodsBase
```

```
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64InternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink>
        ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
        ↪ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
        ↪ linksIndexParts, header) { }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe class UInt64InternalLinksTargetsRecursionlessSizeBalancedTreeMethods :
    ↪ UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public
        ↪ UInt64InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
        ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
        ↪ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
        ↪ linksIndexParts, header) { }

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

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

namespace Platform.Data.Doublets.Memory.Split.Specific
{
    public unsafe class UInt64SplitMemoryLinks : SplitMemoryLinksBase<TLink>
    {
        private readonly Func<ILinksTreeMethods<TLink>> _createInternalSourceTreeMethods;
        private readonly Func<ILinksTreeMethods<TLink>> _createExternalSourceTreeMethods;
        private readonly Func<ILinksTreeMethods<TLink>> _createInternalTargetTreeMethods;
        private readonly Func<ILinksTreeMethods<TLink>> _createExternalTargetTreeMethods;
        private LinksHeader<ulong>* _header;
        private RawLinkDataPart<ulong>* _linksDataParts;
        private RawLinkIndexPart<ulong>* _linksIndexParts;

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
32          public UInt64SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪  indexMemory, long memoryReservationStep, LinksConstants<TLink> constants,
            ↪  IndexTreeType indexTreeType, bool useLinkedList) : base(dataMemory, indexMemory,
            ↪  memoryReservationStep, constants, useLinkedList)
33          {
34              if (indexTreeType == IndexTreeType.SizeBalancedTree)
35              {
36                  _createInternalSourceTreeMethods = () => new
                    ↪  UInt64InternalLinksSourcesSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
37                  _createExternalSourceTreeMethods = () => new
                    ↪  UInt64ExternalLinksSourcesSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
38                  _createInternalTargetTreeMethods = () => new
                    ↪  UInt64InternalLinksTargetsSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
39                  _createExternalTargetTreeMethods = () => new
                    ↪  UInt64ExternalLinksTargetsSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
40              }
41              else
42              {
43                  _createInternalSourceTreeMethods = () => new
                    ↪  UInt64InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
44                  _createExternalSourceTreeMethods = () => new
                    ↪  UInt64ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
45                  _createInternalTargetTreeMethods = () => new
                    ↪  UInt64InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
46                  _createExternalTargetTreeMethods = () => new
                    ↪  UInt64ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(Constants,
                    ↪  _linksDataParts, _linksIndexParts, _header);
47              }
48              Init(dataMemory, indexMemory);
49          }
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override void SetPointers(IResizableDirectMemory dataMemory,
            ↪  IResizableDirectMemory indexMemory)
53          {
54              _linksDataParts = (RawLinkDataPart<TLink>*)dataMemory.Pointer;
55              _linksIndexParts = (RawLinkIndexPart<TLink>*)indexMemory.Pointer;
56              _header = (LinksHeader<TLink>*)indexMemory.Pointer;
57              if (_useLinkedList)
58              {
59                  InternalSourcesListMethods = new
                    ↪  UInt64InternalLinksSourcesLinkedListMethods(Constants, _linksDataParts,
                    ↪  _linksIndexParts);
60              }
61              else
62              {
63                  InternalSourcesTreeMethods = _createInternalSourceTreeMethods();
64              }
65              ExternalSourcesTreeMethods = _createExternalSourceTreeMethods();
66              InternalTargetsTreeMethods = _createInternalTargetTreeMethods();
67              ExternalTargetsTreeMethods = _createExternalTargetTreeMethods();
68              UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_linksDataParts, _header);
69          }
70
71          [MethodImpl(MethodImplOptions.AggressiveInlining)]
72          protected override void ResetPointers()
73          {
74              base.ResetPointers();
75              _linksDataParts = null;
76              _linksIndexParts = null;
77              _header = null;
78          }
79
80          [MethodImpl(MethodImplOptions.AggressiveInlining)]
81          protected override ref LinksHeader<TLink> GetHeaderReference() => ref *_header;
82
83          [MethodImpl(MethodImplOptions.AggressiveInlining)]
84          protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex)
            ↪  => ref _linksDataParts[linkIndex];
85
86          [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
87          protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
            ↪  linkIndex) => ref _linksIndexParts[linkIndex];
88
89          [MethodImpl(MethodImplOptions.AggressiveInlining)]
90          protected override bool AreEqual(ulong first, ulong second) => first == second;
91
92          [MethodImpl(MethodImplOptions.AggressiveInlining)]
93          protected override bool LessThan(ulong first, ulong second) => first < second;
94
95          [MethodImpl(MethodImplOptions.AggressiveInlining)]
96          protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
97
98          [MethodImpl(MethodImplOptions.AggressiveInlining)]
99          protected override bool GreaterThan(ulong first, ulong second) => first > second;
100
101         [MethodImpl(MethodImplOptions.AggressiveInlining)]
102         protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
103
104         [MethodImpl(MethodImplOptions.AggressiveInlining)]
105         protected override ulong GetZero() => 0UL;
106
107         [MethodImpl(MethodImplOptions.AggressiveInlining)]
108         protected override ulong GetOne() => 1UL;
109
110         [MethodImpl(MethodImplOptions.AggressiveInlining)]
111         protected override long ConvertToInt64(ulong value) => (long)value;
112
113         [MethodImpl(MethodImplOptions.AggressiveInlining)]
114         protected override ulong ConvertToAddress(long value) => (ulong)value;
115
116         [MethodImpl(MethodImplOptions.AggressiveInlining)]
117         protected override ulong Add(ulong first, ulong second) => first + second;
118
119         [MethodImpl(MethodImplOptions.AggressiveInlining)]
120         protected override ulong Subtract(ulong first, ulong second) => first - second;
121
122         [MethodImpl(MethodImplOptions.AggressiveInlining)]
123         protected override ulong Increment(ulong link) => ++link;
124
125         [MethodImpl(MethodImplOptions.AggressiveInlining)]
126         protected override ulong Decrement(ulong link) => --link;
127     }
128 }
```

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

```
1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.Memory.Split.Generic;
3  using TLink = System.UInt64;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Memory.Split.Specific
8  {
9      public unsafe class UInt64UnusedLinksListMethods : UnusedLinksListMethods<TLink>
10     {
11         private readonly RawLinkDataPart<ulong>* _links;
12         private readonly LinksHeader<ulong>* _header;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public UInt64UnusedLinksListMethods(RawLinkDataPart<ulong>* links, LinksHeader<ulong>*
           ↪  header)
16             : base((byte*)links, (byte*)header)
17         {
18             _links = links;
19             _header = header;
20         }
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
           ↪  ref _links[link];
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override ref LinksHeader<TLink> GetHeaderReference() => ref *_header;
27     }
28 }
```

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

```
1  using System;
2  using System.Text;
3  using System.Collections.Generic;
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
142         }
143
144         [MethodImpl(MethodImplOptions.AggressiveInlining)]
145         protected virtual void SetBalanceValue(ref TLink storedValue, sbyte value)
146         {
147             unchecked
148             {
149                 var packagedValue = _int32ToAddressConverter.Convert((byte)value >> 5 & 4 |
                    ↪ value & 3);
150                 var modified = Bit<TLink>.PartialWrite(storedValue, packagedValue, 0, 3);
151                 storedValue = modified;
152             }
153         }
154
155         public TLink this[TLink index]
156         {
157             [MethodImpl(MethodImplOptions.AggressiveInlining)]
158             get
159             {
160                 var root = GetTreeRoot();
161                 if (GreaterOrEqualThan(index, GetSize(root)))
162                 {
163                     return Zero;
164                 }
165                 while (!EqualToZero(root))
166                 {
167                     var left = GetLeftOrDefault(root);
168                     var leftSize = GetSizeOrZero(left);
169                     if (LessThan(index, leftSize))
170                     {
171                         root = left;
172                         continue;
173                     }
174                     if (AreEqual(index, leftSize))
175                     {
176                         return root;
177                     }
178                     root = GetRightOrDefault(root);
179                     index = Subtract(index, Increment(leftSize));
180                 }
181                 return Zero; // TODO: Impossible situation exception (only if tree structure
                    ↪ broken)
182             }
183         }
184
185         /// <summary>
186         /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
                ↪ (концом).
187         /// </summary>
188         /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
189         /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
190         /// <returns>Индекс искомой связи.</returns>
191         [MethodImpl(MethodImplOptions.AggressiveInlining)]
192         public TLink Search(TLink source, TLink target)
193         {
194             var root = GetTreeRoot();
195             while (!EqualToZero(root))
196             {
197                 ref var rootLink = ref GetLinkReference(root);
198                 var rootSource = rootLink.Source;
199                 var rootTarget = rootLink.Target;
200                 if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                    ↪ node.Key < root.Key
201                 {
202                     root = GetLeftOrDefault(root);
203                 }
204                 else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
                    ↪ node.Key > root.Key
205                 {
206                     root = GetRightOrDefault(root);
207                 }
208                 else // node.Key == root.Key
209                 {
210                     return root;
211                 }
212             }
213             return Zero;
214         }
215
```

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

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

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

```
            }
            return Continue;
        }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
55
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
58          {
59              ref var firstLink = ref GetLinkReference(first);
60              ref var secondLink = ref GetLinkReference(second);
61              return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
62          }
63
64          [MethodImpl(MethodImplOptions.AggressiveInlining)]
65          protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
66          {
67              ref var firstLink = ref GetLinkReference(first);
68              ref var secondLink = ref GetLinkReference(second);
69              return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
70          }
71
72          public TLink this[TLink index]
73          {
74              [MethodImpl(MethodImplOptions.AggressiveInlining)]
75              get
76              {
77                  var root = GetTreeRoot();
78                  if (GreaterOrEqualThan(index, GetSize(root)))
79                  {
80                      return Zero;
81                  }
82                  while (!EqualToZero(root))
83                  {
84                      var left = GetLeftOrDefault(root);
85                      var leftSize = GetSizeOrZero(left);
86                      if (LessThan(index, leftSize))
87                      {
88                          root = left;
89                          continue;
90                      }
91                      if (AreEqual(index, leftSize))
92                      {
93                          return root;
94                      }
95                      root = GetRightOrDefault(root);
96                      index = Subtract(index, Increment(leftSize));
97                  }
98                  return Zero; // TODO: Impossible situation exception (only if tree structure
                    ↪  broken)
99              }
100         }
101
102         /// <summary>
103         /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
            ↪  (концом).
104         /// </summary>
105         /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
106         /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
107         /// <returns>Индекс искомой связи.</returns>
108         [MethodImpl(MethodImplOptions.AggressiveInlining)]
109         public TLink Search(TLink source, TLink target)
110         {
111             var root = GetTreeRoot();
112             while (!EqualToZero(root))
113             {
114                 ref var rootLink = ref GetLinkReference(root);
115                 var rootSource = rootLink.Source;
116                 var rootTarget = rootLink.Target;
117                 if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                    ↪  node.Key < root.Key
118                 {
119                     root = GetLeftOrDefault(root);
120                 }
121                 else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
                    ↪  node.Key > root.Key
122                 {
123                     root = GetRightOrDefault(root);
124                 }
125                 else // node.Key == root.Key
126                 {
```

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

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

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

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

```
                }
                if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
                {
                    return @break;
                }
                if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
                {
                    return @break;
                }
            }
            return @continue;
        }

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

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

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

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

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

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

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

```
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

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

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

```
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          protected override bool GetLeftIsChild(TLink node) =>
            ↪  GetLeftIsChildValue(GetLinkReference(node).SizeAsSource);
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          protected override void SetLeftIsChild(TLink node, bool value) =>
            ↪  SetLeftIsChildValue(ref GetLinkReference(node).SizeAsSource, value);
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          protected override bool GetRightIsChild(TLink node) =>
            ↪  GetRightIsChildValue(GetLinkReference(node).SizeAsSource);
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected override void SetRightIsChild(TLink node, bool value) =>
            ↪  SetRightIsChildValue(ref GetLinkReference(node).SizeAsSource, value);
47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          protected override sbyte GetBalance(TLink node) =>
            ↪  GetBalanceValue(GetLinkReference(node).SizeAsSource);
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
            ↪  GetLinkReference(node).SizeAsSource, value);
53
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          protected override TLink GetTreeRoot() => GetHeaderReference().RootAsSource;
56
57          [MethodImpl(MethodImplOptions.AggressiveInlining)]
58          protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;
59
60          [MethodImpl(MethodImplOptions.AggressiveInlining)]
61          protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
            ↪  TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
            ↪  (AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
62
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
            ↪  TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
            ↪  (AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
65
66          [MethodImpl(MethodImplOptions.AggressiveInlining)]
67          protected override void ClearNode(TLink node)
68          {
69              ref var link = ref GetLinkReference(node);
70              link.LeftAsSource = Zero;
71              link.RightAsSource = Zero;
72              link.SizeAsSource = Zero;
73          }
74      }
75  }
```

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

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

```
        protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsSource;

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLink node, TLink size) =>
        ↪  GetLinkReference(node).SizeAsSource = size;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetTreeRoot() => GetHeaderReference().RootAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
        ↪  TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
        ↪  (AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
        ↪  TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
        ↪  (AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLink node)
        {
            ref var link = ref GetLinkReference(node);
            link.LeftAsSource = Zero;
            link.RightAsSource = Zero;
            link.SizeAsSource = Zero;
        }
    }
}
```

1.82 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesSizeBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Generic
{
    public unsafe class LinksSourcesSizeBalancedTreeMethods<TLink> :
    ↪  LinksSizeBalancedTreeMethodsBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
        ↪  byte* header) : base(constants, links, header) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLink GetLeftReference(TLink node) => ref
        ↪  GetLinkReference(node).LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLink GetRightReference(TLink node) => ref
        ↪  GetLinkReference(node).RightAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(TLink node, TLink left) =>
        ↪  GetLinkReference(node).LeftAsSource = left;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(TLink node, TLink right) =>
        ↪  GetLinkReference(node).RightAsSource = right;
```

```

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLink node, TLink size) =>
        ↪  GetLinkReference(node).SizeAsSource = size;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetTreeRoot() => GetHeaderReference().RootAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
        ↪  TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
        ↪  (AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
        ↪  TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
        ↪  (AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLink node)
        {
            ref var link = ref GetLinkReference(node);
            link.LeftAsSource = Zero;
            link.RightAsSource = Zero;
            link.SizeAsSource = Zero;
        }
    }
}
```

1.83 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsAvlBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Generic
{
    public unsafe class LinksTargetsAvlBalancedTreeMethods<TLink> :
    ↪  LinksAvlBalancedTreeMethodsBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksTargetsAvlBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
        ↪  byte* header) : base(constants, links, header) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLink GetLeftReference(TLink node) => ref
        ↪  GetLinkReference(node).LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLink GetRightReference(TLink node) => ref
        ↪  GetLinkReference(node).RightAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(TLink node, TLink left) =>
        ↪  GetLinkReference(node).LeftAsTarget = left;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(TLink node, TLink right) =>
        ↪  GetLinkReference(node).RightAsTarget = right;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetSize(TLink node) =>
        ↪  GetSizeValue(GetLinkReference(node).SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
        ↪  GetLinkReference(node).SizeAsTarget, size);

```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetLeftIsChild(TLink node) =>
        ↪ GetLeftIsChildValue(GetLinkReference(node).SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeftIsChild(TLink node, bool value) =>
        ↪ SetLeftIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetRightIsChild(TLink node) =>
        ↪ GetRightIsChildValue(GetLinkReference(node).SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRightIsChild(TLink node, bool value) =>
        ↪ SetRightIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override sbyte GetBalance(TLink node) =>
        ↪ GetBalanceValue(GetLinkReference(node).SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
        ↪ GetLinkReference(node).SizeAsTarget, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetTreeRoot() => GetHeaderReference().RootAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
        ↪ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
        ↪ (AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
        ↪ TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
        ↪ (AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLink node)
        {
            ref var link = ref GetLinkReference(node);
            link.LeftAsTarget = Zero;
            link.RightAsTarget = Zero;
            link.SizeAsTarget = Zero;
        }
    }
}
```

1.84 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsRecursionlessSizeBalancedTreeMetho

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Generic
{
    public unsafe class LinksTargetsRecursionlessSizeBalancedTreeMethods<TLink> :
    ↪ LinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink> constants,
        ↪ byte* links, byte* header) : base(constants, links, header) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLink GetLeftReference(TLink node) => ref
        ↪ GetLinkReference(node).LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLink GetRightReference(TLink node) => ref
        ↪ GetLinkReference(node).RightAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsTarget;
```

```

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(TLink node, TLink left) =>
        ↪ GetLinkReference(node).LeftAsTarget = left;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(TLink node, TLink right) =>
        ↪ GetLinkReference(node).RightAsTarget = right;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLink node, TLink size) =>
        ↪ GetLinkReference(node).SizeAsTarget = size;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetTreeRoot() => GetHeaderReference().RootAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
        ↪ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
        ↪ (AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
        ↪ TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
        ↪ (AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLink node)
        {
            ref var link = ref GetLinkReference(node);
            link.LeftAsTarget = Zero;
            link.RightAsTarget = Zero;
            link.SizeAsTarget = Zero;
        }
    }
}
```

1.85 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsSizeBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Generic
{
    public unsafe class LinksTargetsSizeBalancedTreeMethods<TLink> :
    ↪ LinksSizeBalancedTreeMethodsBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
        ↪ byte* header) : base(constants, links, header) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLink GetLeftReference(TLink node) => ref
        ↪ GetLinkReference(node).LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref TLink GetRightReference(TLink node) => ref
        ↪ GetLinkReference(node).RightAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(TLink node, TLink left) =>
        ↪ GetLinkReference(node).LeftAsTarget = left;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(TLink node, TLink right) =>
        ↪ GetLinkReference(node).RightAsTarget = right;

```

```
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsTarget;
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          protected override void SetSize(TLink node, TLink size) =>
            ↪  GetLinkReference(node).SizeAsTarget = size;
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          protected override TLink GetTreeRoot() => GetHeaderReference().RootAsTarget;
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
            ↪  TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
            ↪  (AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource));
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
            ↪  TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
            ↪  (AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource));
47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          protected override void ClearNode(TLink node)
50          {
51              ref var link = ref GetLinkReference(node);
52              link.LeftAsTarget = Zero;
53              link.RightAsTarget = Zero;
54              link.SizeAsTarget = Zero;
55          }
56      }
57  }
```

1.86 ./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      public unsafe class UnitedMemoryLinks<TLink> : UnitedMemoryLinksBase<TLink>
12      {
13          private readonly Func<ILinksTreeMethods<TLink>> _createSourceTreeMethods;
14          private readonly Func<ILinksTreeMethods<TLink>> _createTargetTreeMethods;
15          private byte* _header;
16          private byte* _links;
17
18          [MethodImpl(MethodImplOptions.AggressiveInlining)]
19          public UnitedMemoryLinks(string address) : this(address, DefaultLinksSizeStep) { }
20
21          /// <summary>
22          /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
            ↪  минимальным шагом расширения базы данных.
23          /// </summary>
24          /// <param name="address">Полный пусть к файлу базы данных.</param>
25          /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
            ↪  байтах.</param>
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          public UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
            ↪  FileMappedResizableDirectMemory(address, memoryReservationStep),
            ↪  memoryReservationStep) { }
28
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          public UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
            ↪  DefaultLinksSizeStep) { }
31
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          public UnitedMemoryLinks(IResizableDirectMemory memory, long memoryReservationStep) :
            ↪  this(memory, memoryReservationStep, Default<LinksConstants<TLink>>.Instance,
            ↪  IndexTreeType.Default) { }
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
36          public UnitedMemoryLinks(IResizableDirectMemory memory, long memoryReservationStep,
            ↪  LinksConstants<TLink> constants, IndexTreeType indexTreeType) : base(memory,
            ↪  memoryReservationStep, constants)
37          {
38              if (indexTreeType == IndexTreeType.SizedAndThreadedAVLBalancedTree)
39              {
40                  _createSourceTreeMethods = () => new
                    ↪  LinksSourcesAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
41                  _createTargetTreeMethods = () => new
                    ↪  LinksTargetsAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
42              }
43              else
44              {
45                  _createSourceTreeMethods = () => new
                    ↪  LinksSourcesSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
46                  _createTargetTreeMethods = () => new
                    ↪  LinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
47              }
48              Init(memory, memoryReservationStep);
49          }
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override void SetPointers(IResizableDirectMemory memory)
53          {
54              _links = (byte*)memory.Pointer;
55              _header = _links;
56              SourcesTreeMethods = _createSourceTreeMethods();
57              TargetsTreeMethods = _createTargetTreeMethods();
58              UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_links, _header);
59          }
60
61          [MethodImpl(MethodImplOptions.AggressiveInlining)]
62          protected override void ResetPointers()
63          {
64              base.ResetPointers();
65              _links = null;
66              _header = null;
67          }
68
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          protected override ref LinksHeader<TLink> GetHeaderReference() => ref
            ↪  AsRef<LinksHeader<TLink>>(_header);
71
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          protected override ref RawLink<TLink> GetLinkReference(TLink linkIndex) => ref
            ↪  AsRef<RawLink<TLink>>(_links + (LinkSizeInBytes * ConvertToInt64(linkIndex)));
74      }
75  }
```

1.87 ./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
11  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13  namespace Platform.Data.Doublets.Memory.United.Generic
14  {
15      public abstract class UnitedMemoryLinksBase<TLink> : DisposableBase, ILinks<TLink>
16      {
17          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
18          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
19          private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
            ↪  UncheckedConverter<TLink, long>.Default;
20          private static readonly UncheckedConverter<long, TLink> _int64ToAddressConverter =
            ↪  UncheckedConverter<long, TLink>.Default;
21
22          private static readonly TLink _zero = default;
23          private static readonly TLink _one = Arithmetic.Increment(_zero);
24
25          /// <summary>Возвращает размер одной связи в байтах.</summary>
26          /// <remarks>
```

```
        /// Используется только во вне класса, не рекомедуется использовать внутри.
        /// Так как во вне не обязательно будет доступен unsafe C#.
        /// </remarks>
        public static readonly long LinkSizeInBytes = RawLink<TLink>.SizeInBytes;

        public static readonly long LinkHeaderSizeInBytes = LinksHeader<TLink>.SizeInBytes;

        public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;

        protected readonly IResizableDirectMemory _memory;
        protected readonly long _memoryReservationStep;

        protected ILinksTreeMethods<TLink> TargetsTreeMethods;
        protected ILinksTreeMethods<TLink> SourcesTreeMethods;
        // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
        ↪  нужно использовать не список а дерево, так как так можно быстрее проверить на
        ↪  наличие связи внутри
        protected ILinksListMethods<TLink> UnusedLinksListMethods;

        /// <summary>
        /// Возвращает общее число связей находящихся в хранилище.
        /// </summary>
        protected virtual TLink Total
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get
            {
                ref var header = ref GetHeaderReference();
                return Subtract(header.AllocatedLinks, header.FreeLinks);
            }
        }

        public virtual LinksConstants<TLink> Constants
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected UnitedMemoryLinksBase(IResizableDirectMemory memory, long
        ↪  memoryReservationStep, LinksConstants<TLink> constants)
        {
            _memory = memory;
            _memoryReservationStep = memoryReservationStep;
            Constants = constants;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected UnitedMemoryLinksBase(IResizableDirectMemory memory, long
        ↪  memoryReservationStep) : this(memory, memoryReservationStep,
        ↪  Default<LinksConstants<TLink>>.Instance) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual void Init(IResizableDirectMemory memory, long memoryReservationStep)
        {
            if (memory.ReservedCapacity < memoryReservationStep)
            {
                memory.ReservedCapacity = memoryReservationStep;
            }
            SetPointers(memory);
            ref var header = ref GetHeaderReference();
            // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
            memory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) * LinkSizeInBytes) +
            ↪  LinkHeaderSizeInBytes;
            // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
            header.ReservedLinks = ConvertToAddress((memory.ReservedCapacity -
            ↪  LinkHeaderSizeInBytes) / LinkSizeInBytes);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Count(IList<TLink> restrictions)
        {
            // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
            if (restrictions.Count == 0)
            {
                return Total;
            }
            var constants = Constants;
            var any = constants.Any;
            var index = restrictions[constants.IndexPart];
```

```
100             if (restrictions.Count == 1)
101             {
102                 if (AreEqual(index, any))
103                 {
104                     return Total;
105                 }
106                 return Exists(index) ? GetOne() : GetZero();
107             }
108             if (restrictions.Count == 2)
109             {
110                 var value = restrictions[1];
111                 if (AreEqual(index, any))
112                 {
113                     if (AreEqual(value, any))
114                     {
115                         return Total; // Any - как отсутствие ограничения
116                     }
117                     return Add(SourcesTreeMethods.CountUsages(value),
                        ↪ TargetsTreeMethods.CountUsages(value));
118                 }
119                 else
120                 {
121                     if (!Exists(index))
122                     {
123                         return GetZero();
124                     }
125                     if (AreEqual(value, any))
126                     {
127                         return GetOne();
128                     }
129                     ref var storedLinkValue = ref GetLinkReference(index);
130                     if (AreEqual(storedLinkValue.Source, value) ||
                        ↪ AreEqual(storedLinkValue.Target, value))
131                     {
132                         return GetOne();
133                     }
134                     return GetZero();
135                 }
136             }
137             if (restrictions.Count == 3)
138             {
139                 var source = restrictions[constants.SourcePart];
140                 var target = restrictions[constants.TargetPart];
141                 if (AreEqual(index, any))
142                 {
143                     if (AreEqual(source, any) && AreEqual(target, any))
144                     {
145                         return Total;
146                     }
147                     else if (AreEqual(source, any))
148                     {
149                         return TargetsTreeMethods.CountUsages(target);
150                     }
151                     else if (AreEqual(target, any))
152                     {
153                         return SourcesTreeMethods.CountUsages(source);
154                     }
155                     else //if(source != Any && target != Any)
156                     {
157                         // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
158                         var link = SourcesTreeMethods.Search(source, target);
159                         return AreEqual(link, constants.Null) ? GetZero() : GetOne();
160                     }
161                 }
162                 else
163                 {
164                     if (!Exists(index))
165                     {
166                         return GetZero();
167                     }
168                     if (AreEqual(source, any) && AreEqual(target, any))
169                     {
170                         return GetOne();
171                     }
172                     ref var storedLinkValue = ref GetLinkReference(index);
173                     if (!AreEqual(source, any) && !AreEqual(target, any))
174                     {
```

```
                        if (AreEqual(storedLinkValue.Source, source) &&
                        ↪  AreEqual(storedLinkValue.Target, target))
                        {
                            return GetOne();
                        }
                        return GetZero();
                    }
                    var value = default(TLink);
                    if (AreEqual(source, any))
                    {
                        value = target;
                    }
                    if (AreEqual(target, any))
                    {
                        value = source;
                    }
                    if (AreEqual(storedLinkValue.Source, value) ||
                    ↪  AreEqual(storedLinkValue.Target, value))
                    {
                        return GetOne();
                    }
                    return GetZero();
                }
            }
            throw new NotSupportedException("Другие размеры и способы ограничений не
            ↪  поддерживаются.");
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
        {
            var constants = Constants;
            var @break = constants.Break;
            if (restrictions.Count == 0)
            {
                for (var link = GetOne(); LessOrEqualThan(link,
                ↪  GetHeaderReference().AllocatedLinks); link = Increment(link))
                {
                    if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
                    {
                        return @break;
                    }
                }
                return @break;
            }
            var @continue = constants.Continue;
            var any = constants.Any;
            var index = restrictions[constants.IndexPart];
            if (restrictions.Count == 1)
            {
                if (AreEqual(index, any))
                {
                    return Each(handler, Array.Empty<TLink>());
                }
                if (!Exists(index))
                {
                    return @continue;
                }
                return handler(GetLinkStruct(index));
            }
            if (restrictions.Count == 2)
            {
                var value = restrictions[1];
                if (AreEqual(index, any))
                {
                    if (AreEqual(value, any))
                    {
                        return Each(handler, Array.Empty<TLink>());
                    }
                    if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
                    {
                        return @break;
                    }
                    return Each(handler, new Link<TLink>(index, any, value));
                }
                else
                {
                    if (!Exists(index))
```

```
249                     {
250                         return @continue;
251                     }
252                     if (AreEqual(value, any))
253                     {
254                         return handler(GetLinkStruct(index));
255                     }
256                     ref var storedLinkValue = ref GetLinkReference(index);
257                     if (AreEqual(storedLinkValue.Source, value) ||
258                         AreEqual(storedLinkValue.Target, value))
259                     {
260                         return handler(GetLinkStruct(index));
261                     }
262                     return @continue;
263                 }
264             }
265             if (restrictions.Count == 3)
266             {
267                 var source = restrictions[constants.SourcePart];
268                 var target = restrictions[constants.TargetPart];
269                 if (AreEqual(index, any))
270                 {
271                     if (AreEqual(source, any) && AreEqual(target, any))
272                     {
273                         return Each(handler, Array.Empty<TLink>());
274                     }
275                     else if (AreEqual(source, any))
276                     {
277                         return TargetsTreeMethods.EachUsage(target, handler);
278                     }
279                     else if (AreEqual(target, any))
280                     {
281                         return SourcesTreeMethods.EachUsage(source, handler);
282                     }
283                     else //if(source != Any && target != Any)
284                     {
285                         var link = SourcesTreeMethods.Search(source, target);
286                         return AreEqual(link, constants.Null) ? @continue :
                            ↪  handler(GetLinkStruct(link));
287                     }
288                 }
289                 else
290                 {
291                     if (!Exists(index))
292                     {
293                         return @continue;
294                     }
295                     if (AreEqual(source, any) && AreEqual(target, any))
296                     {
297                         return handler(GetLinkStruct(index));
298                     }
299                     ref var storedLinkValue = ref GetLinkReference(index);
300                     if (!AreEqual(source, any) && !AreEqual(target, any))
301                     {
302                         if (AreEqual(storedLinkValue.Source, source) &&
303                             AreEqual(storedLinkValue.Target, target))
304                         {
305                             return handler(GetLinkStruct(index));
306                         }
307                         return @continue;
308                     }
309                     var value = default(TLink);
310                     if (AreEqual(source, any))
311                     {
312                         value = target;
313                     }
314                     if (AreEqual(target, any))
315                     {
316                         value = source;
317                     }
318                     if (AreEqual(storedLinkValue.Source, value) ||
319                         AreEqual(storedLinkValue.Target, value))
320                     {
321                         return handler(GetLinkStruct(index));
322                     }
323                     return @continue;
324                 }
325             }
```

```
326             throw new NotSupportedException("Другие размеры и способы ограничений не
                ↪ поддерживаются.");
327         }
328 
329         /// <remarks>
330         /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
                ↪ в другом месте (но не в менеджере памяти, а в логике Links)
331         /// </remarks>
332         [MethodImpl(MethodImplOptions.AggressiveInlining)]
333         public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
334         {
335             var constants = Constants;
336             var @null = constants.Null;
337             var linkIndex = restrictions[constants.IndexPart];
338             ref var link = ref GetLinkReference(linkIndex);
339             ref var header = ref GetHeaderReference();
340             ref var firstAsSource = ref header.RootAsSource;
341             ref var firstAsTarget = ref header.RootAsTarget;
342             // Будет корректно работать только в том случае, если пространство выделенной связи
                ↪ предварительно заполнено нулями
343             if (!AreEqual(link.Source, @null))
344             {
345                 SourcesTreeMethods.Detach(ref firstAsSource, linkIndex);
346             }
347             if (!AreEqual(link.Target, @null))
348             {
349                 TargetsTreeMethods.Detach(ref firstAsTarget, linkIndex);
350             }
351             link.Source = substitution[constants.SourcePart];
352             link.Target = substitution[constants.TargetPart];
353             if (!AreEqual(link.Source, @null))
354             {
355                 SourcesTreeMethods.Attach(ref firstAsSource, linkIndex);
356             }
357             if (!AreEqual(link.Target, @null))
358             {
359                 TargetsTreeMethods.Attach(ref firstAsTarget, linkIndex);
360             }
361             return linkIndex;
362         }
363 
364         /// <remarks>
365         /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
                ↪ пространство
366         /// </remarks>
367         [MethodImpl(MethodImplOptions.AggressiveInlining)]
368         public virtual TLink Create(IList<TLink> restrictions)
369         {
370             ref var header = ref GetHeaderReference();
371             var freeLink = header.FirstFreeLink;
372             if (!AreEqual(freeLink, Constants.Null))
373             {
374                 UnusedLinksListMethods.Detach(freeLink);
375             }
376             else
377             {
378                 var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
379                 if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
380                 {
381                     throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
382                 }
383                 if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
384                 {
385                     _memory.ReservedCapacity += _memoryReservationStep;
386                     SetPointers(_memory);
387                     header = ref GetHeaderReference();
388                     header.ReservedLinks = ConvertToAddress(_memory.ReservedCapacity /
                        ↪ LinkSizeInBytes);
389                 }
390                 freeLink = header.AllocatedLinks = Increment(header.AllocatedLinks);
391                 _memory.UsedCapacity += LinkSizeInBytes;
392             }
393             return freeLink;
394         }
395 
396         [MethodImpl(MethodImplOptions.AggressiveInlining)]
397         public virtual void Delete(IList<TLink> restrictions)
398         {
399             ref var header = ref GetHeaderReference();
```

```
400             var link = restrictions[Constants.IndexPart];
401             if (LessThan(link, header.AllocatedLinks))
402             {
403                 UnusedLinksListMethods.AttachAsFirst(link);
404             }
405             else if (AreEqual(link, header.AllocatedLinks))
406             {
407                 header.AllocatedLinks = Decrement(header.AllocatedLinks);
408                 _memory.UsedCapacity -= LinkSizeInBytes;
409                 // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
                    ↪ пока не дойдём до первой существующей связи
410                 // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
411                 while (GreaterThan(header.AllocatedLinks, GetZero()) &&
                    ↪ IsUnusedLink(header.AllocatedLinks))
412                 {
413                     UnusedLinksListMethods.Detach(header.AllocatedLinks);
414                     header.AllocatedLinks = Decrement(header.AllocatedLinks);
415                     _memory.UsedCapacity -= LinkSizeInBytes;
416                 }
417             }
418         }
419 
420         [MethodImpl(MethodImplOptions.AggressiveInlining)]
421         public IList<TLink> GetLinkStruct(TLink linkIndex)
422         {
423             ref var link = ref GetLinkReference(linkIndex);
424             return new Link<TLink>(linkIndex, link.Source, link.Target);
425         }
426 
427         /// <remarks>
428         /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
                ↪ адрес реально поменялся
429         ///
430         /// Указатель this.links может быть в том же месте,
431         /// так как 0-я связь не используется и имеет такой же размер как Header,
432         /// поэтому header размещается в том же месте, что и 0-я связь
433         /// </remarks>
434         [MethodImpl(MethodImplOptions.AggressiveInlining)]
435         protected abstract void SetPointers(IResizableDirectMemory memory);
436 
437         [MethodImpl(MethodImplOptions.AggressiveInlining)]
438         protected virtual void ResetPointers()
439         {
440             SourcesTreeMethods = null;
441             TargetsTreeMethods = null;
442             UnusedLinksListMethods = null;
443         }
444 
445         [MethodImpl(MethodImplOptions.AggressiveInlining)]
446         protected abstract ref LinksHeader<TLink> GetHeaderReference();
447 
448         [MethodImpl(MethodImplOptions.AggressiveInlining)]
449         protected abstract ref RawLink<TLink> GetLinkReference(TLink linkIndex);
450 
451         [MethodImpl(MethodImplOptions.AggressiveInlining)]
452         protected virtual bool Exists(TLink link)
453             => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
454             && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
455             && !IsUnusedLink(link);
456 
457         [MethodImpl(MethodImplOptions.AggressiveInlining)]
458         protected virtual bool IsUnusedLink(TLink linkIndex)
459         {
460             if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
                    ↪ is not needed
461             {
462                 ref var link = ref GetLinkReference(linkIndex);
463                 return AreEqual(link.SizeAsSource, default) && !AreEqual(link.Source, default);
464             }
465             else
466             {
467                 return true;
468             }
469         }
470 
471         [MethodImpl(MethodImplOptions.AggressiveInlining)]
472         protected virtual TLink GetOne() => _one;
473 
474         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
475         protected virtual TLink GetZero() => default;
476
477         [MethodImpl(MethodImplOptions.AggressiveInlining)]
478         protected virtual bool AreEqual(TLink first, TLink second) =>
            ↪ _equalityComparer.Equals(first, second);
479
480         [MethodImpl(MethodImplOptions.AggressiveInlining)]
481         protected virtual bool LessThan(TLink first, TLink second) => _comparer.Compare(first,
            ↪ second) < 0;
482
483         [MethodImpl(MethodImplOptions.AggressiveInlining)]
484         protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
            ↪ _comparer.Compare(first, second) <= 0;
485
486         [MethodImpl(MethodImplOptions.AggressiveInlining)]
487         protected virtual bool GreaterThan(TLink first, TLink second) =>
            ↪ _comparer.Compare(first, second) > 0;
488
489         [MethodImpl(MethodImplOptions.AggressiveInlining)]
490         protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
            ↪ _comparer.Compare(first, second) >= 0;
491
492         [MethodImpl(MethodImplOptions.AggressiveInlining)]
493         protected virtual long ConvertToInt64(TLink value) =>
            ↪ _addressToInt64Converter.Convert(value);
494
495         [MethodImpl(MethodImplOptions.AggressiveInlining)]
496         protected virtual TLink ConvertToAddress(long value) =>
            ↪ _int64ToAddressConverter.Convert(value);
497
498         [MethodImpl(MethodImplOptions.AggressiveInlining)]
499         protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
            ↪ second);
500
501         [MethodImpl(MethodImplOptions.AggressiveInlining)]
502         protected virtual TLink Subtract(TLink first, TLink second) =>
            ↪ Arithmetic<TLink>.Subtract(first, second);
503
504         [MethodImpl(MethodImplOptions.AggressiveInlining)]
505         protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);
506
507         [MethodImpl(MethodImplOptions.AggressiveInlining)]
508         protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);
509
510         #region Disposable
511
512         protected override bool AllowMultipleDisposeCalls
513         {
514             [MethodImpl(MethodImplOptions.AggressiveInlining)]
515             get => true;
516         }
517
518         [MethodImpl(MethodImplOptions.AggressiveInlining)]
519         protected override void Dispose(bool manual, bool wasDisposed)
520         {
521             if (!wasDisposed)
522             {
523                 ResetPointers();
524                 _memory.DisposeIfPossible();
525             }
526         }
527
528         #endregion
529     }
530 }
```

1.88 ./csharp/Platform.Data.Doublets/Memory/United/Generic/UnusedLinksListMethods.cs

```
1  using System.Runtime.CompilerServices;
2  using Platform.Collections.Methods.Lists;
3  using Platform.Converters;
4  using static System.Runtime.CompilerServices.Unsafe;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Memory.United.Generic
9  {
10     public unsafe class UnusedLinksListMethods<TLink> :
       ↪ AbsoluteCircularDoublyLinkedListMethods<TLink>, ILinksListMethods<TLink>
11     {
```

```
12          private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
            ↪  UncheckedConverter<TLink, long>.Default;
13
14          private readonly byte* _links;
15          private readonly byte* _header;
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          public UnusedLinksListMethods(byte* links, byte* header)
19          {
20              _links = links;
21              _header = header;
22          }
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
            ↪  AsRef<LinksHeader<TLink>>(_header);
26
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
            ↪  AsRef<RawLink<TLink>>(_links + (RawLink<TLink>.SizeInBytes *
            ↪  _addressToInt64Converter.Convert(link)));
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          protected override TLink GetFirst() => GetHeaderReference().FirstFreeLink;
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          protected override TLink GetLast() => GetHeaderReference().LastFreeLink;
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          protected override TLink GetPrevious(TLink element) => GetLinkReference(element).Source;
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          protected override TLink GetNext(TLink element) => GetLinkReference(element).Target;
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          protected override TLink GetSize() => GetHeaderReference().FreeLinks;
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected override void SetFirst(TLink element) => GetHeaderReference().FirstFreeLink =
            ↪  element;
47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          protected override void SetLast(TLink element) => GetHeaderReference().LastFreeLink =
            ↪  element;
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override void SetPrevious(TLink element, TLink previous) =>
            ↪  GetLinkReference(element).Source = previous;
53
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          protected override void SetNext(TLink element, TLink next) =>
            ↪  GetLinkReference(element).Target = next;
56
57          [MethodImpl(MethodImplOptions.AggressiveInlining)]
58          protected override void SetSize(TLink size) => GetHeaderReference().FreeLinks = size;
59      }
60  }
```

1.89 ./csharp/Platform.Data.Doublets/Memory/United/RawLink.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.United
9   {
10      public struct RawLink<TLink> : IEquatable<RawLink<TLink>>
11      {
12          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
13
14          public static readonly long SizeInBytes = Structure<RawLink<TLink>>.Size;
15
16          public TLink Source;
17          public TLink Target;
18          public TLink LeftAsSource;
19          public TLink RightAsSource;
20          public TLink SizeAsSource;
```

```
        public TLink LeftAsTarget;
        public TLink RightAsTarget;
        public TLink SizeAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Equals(object obj) => obj is RawLink<TLink> link ? Equals(link) :
        ↪  false;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Equals(RawLink<TLink> other)
            => _equalityComparer.Equals(Source, other.Source)
            && _equalityComparer.Equals(Target, other.Target)
            && _equalityComparer.Equals(LeftAsSource, other.LeftAsSource)
            && _equalityComparer.Equals(RightAsSource, other.RightAsSource)
            && _equalityComparer.Equals(SizeAsSource, other.SizeAsSource)
            && _equalityComparer.Equals(LeftAsTarget, other.LeftAsTarget)
            && _equalityComparer.Equals(RightAsTarget, other.RightAsTarget)
            && _equalityComparer.Equals(SizeAsTarget, other.SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override int GetHashCode() => (Source, Target, LeftAsSource, RightAsSource,
        ↪  SizeAsSource, LeftAsTarget, RightAsTarget, SizeAsTarget).GetHashCode();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator ==(RawLink<TLink> left, RawLink<TLink> right) =>
        ↪  left.Equals(right);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator !=(RawLink<TLink> left, RawLink<TLink> right) => !(left ==
        ↪  right);
    }
}
```

1.90 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksRecursionlessSizeBalancedTreeMetho

```
using System.Runtime.CompilerServices;
using Platform.Data.Doublets.Memory.United.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Specific
{
    public unsafe abstract class UInt32LinksRecursionlessSizeBalancedTreeMethodsBase :
    ↪  LinksRecursionlessSizeBalancedTreeMethodsBase<uint>
    {
        protected new readonly RawLink<uint>* Links;
        protected new readonly LinksHeader<uint>* Header;

        protected UInt32LinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<uint>
        ↪  constants, RawLink<uint>* links, LinksHeader<uint>* header)
            : base(constants, (byte*)links, (byte*)header)
        {
            Links = links;
            Header = header;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetZero() => 0U;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool EqualToZero(uint value) => value == 0U;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool AreEqual(uint first, uint second) => first == second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThanZero(uint value) => value > 0U;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThan(uint first, uint second) => first > second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterOrEqualThan(uint first, uint second) => first >= second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterOrEqualThanZero(uint value) => true; // value >= 0 is
        ↪  always true for uint

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessOrEqualThanZero(uint value) => value == 0U; // value is
        ↪  always >= 0 for uint
```

```
43
44          [MethodImpl(MethodImplOptions.AggressiveInlining)]
45          protected override bool LessOrEqualThan(uint first, uint second) => first <= second;
46
47          [MethodImpl(MethodImplOptions.AggressiveInlining)]
48          protected override bool LessThanZero(uint value) => false; // value < 0 is always false
            ↪  for uint
49
50          [MethodImpl(MethodImplOptions.AggressiveInlining)]
51          protected override bool LessThan(uint first, uint second) => first < second;
52
53          [MethodImpl(MethodImplOptions.AggressiveInlining)]
54          protected override uint Increment(uint value) => ++value;
55
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          protected override uint Decrement(uint value) => --value;
58
59          [MethodImpl(MethodImplOptions.AggressiveInlining)]
60          protected override uint Add(uint first, uint second) => first + second;
61
62          [MethodImpl(MethodImplOptions.AggressiveInlining)]
63          protected override uint Subtract(uint first, uint second) => first - second;
64
65          [MethodImpl(MethodImplOptions.AggressiveInlining)]
66          protected override bool FirstIsToTheLeftOfSecond(uint first, uint second)
67          {
68              ref var firstLink = ref Links[first];
69              ref var secondLink = ref Links[second];
70              return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
71          }
72
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          protected override bool FirstIsToTheRightOfSecond(uint first, uint second)
75          {
76              ref var firstLink = ref Links[first];
77              ref var secondLink = ref Links[second];
78              return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
79          }
80
81          [MethodImpl(MethodImplOptions.AggressiveInlining)]
82          protected override ref LinksHeader<uint> GetHeaderReference() => ref *Header;
83
84          [MethodImpl(MethodImplOptions.AggressiveInlining)]
85          protected override ref RawLink<uint> GetLinkReference(uint link) => ref Links[link];
86      }
87  }
```

1.91 ./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       public unsafe abstract class UInt32LinksSizeBalancedTreeMethodsBase :
        ↪  LinksSizeBalancedTreeMethodsBase<uint>
9       {
10          protected new readonly RawLink<uint>* Links;
11          protected new readonly LinksHeader<uint>* Header;
12
13          protected UInt32LinksSizeBalancedTreeMethodsBase(LinksConstants<uint> constants,
            ↪  RawLink<uint>* links, LinksHeader<uint>* header)
14              : base(constants, (byte*)links, (byte*)header)
15          {
16              Links = links;
17              Header = header;
18          }
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          protected override uint GetZero() => 0U;
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          protected override bool EqualToZero(uint value) => value == 0U;
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          protected override bool AreEqual(uint first, uint second) => first == second;
28
```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThanZero(uint value) => value > 0U;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThan(uint first, uint second) => first > second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterOrEqualThan(uint first, uint second) => first >= second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterOrEqualThanZero(uint value) => true; // value >= 0 is
        ↪  always true for uint

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessOrEqualThanZero(uint value) => value == 0U; // value is
        ↪  always >= 0 for uint

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessOrEqualThan(uint first, uint second) => first <= second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessThanZero(uint value) => false; // value < 0 is always false
        ↪  for uint

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessThan(uint first, uint second) => first < second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint Increment(uint value) => ++value;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint Decrement(uint value) => --value;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint Add(uint first, uint second) => first + second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint Subtract(uint first, uint second) => first - second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(uint first, uint second)
        {
            ref var firstLink = ref Links[first];
            ref var secondLink = ref Links[second];
            return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
            ↪  secondLink.Source, secondLink.Target);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(uint first, uint second)
        {
            ref var firstLink = ref Links[first];
            ref var secondLink = ref Links[second];
            return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
            ↪  secondLink.Source, secondLink.Target);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref LinksHeader<uint> GetHeaderReference() => ref *Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLink<uint> GetLinkReference(uint link) => ref Links[link];
    }
}
```

1.92 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksSourcesRecursionlessSizeBalancedTre

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Specific
{
    public unsafe class UInt32LinksSourcesRecursionlessSizeBalancedTreeMethods :
    ↪  UInt32LinksRecursionlessSizeBalancedTreeMethodsBase
    {
        public UInt32LinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<uint>
        ↪  constants, RawLink<uint>* links, LinksHeader<uint>* header) : base(constants, links,
        ↪  header) { }

```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref uint GetLeftReference(uint node) => ref Links[node].LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref uint GetRightReference(uint node) => ref
        ↪  Links[node].RightAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetLeft(uint node) => Links[node].LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetRight(uint node) => Links[node].RightAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(uint node, uint left) => Links[node].LeftAsSource = left;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(uint node, uint right) => Links[node].RightAsSource =
        ↪  right;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetSize(uint node) => Links[node].SizeAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(uint node, uint size) => Links[node].SizeAsSource = size;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetTreeRoot() => Header->RootAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetBasePartValue(uint link) => Links[link].Source;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(uint firstSource, uint firstTarget,
        ↪  uint secondSource, uint secondTarget)
            => firstSource < secondSource || (firstSource == secondSource && firstTarget <
            ↪  secondTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(uint firstSource, uint firstTarget,
        ↪  uint secondSource, uint secondTarget)
            => firstSource > secondSource || (firstSource == secondSource && firstTarget >
            ↪  secondTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(uint node)
        {
            ref var link = ref Links[node];
            link.LeftAsSource = 0U;
            link.RightAsSource = 0U;
            link.SizeAsSource = 0U;
        }
    }
}
```

1.93 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksSourcesSizeBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Specific
{
    public unsafe class UInt32LinksSourcesSizeBalancedTreeMethods :
    ↪  UInt32LinksSizeBalancedTreeMethodsBase
    {
        public UInt32LinksSourcesSizeBalancedTreeMethods(LinksConstants<uint> constants,
        ↪  RawLink<uint>* links, LinksHeader<uint>* header) : base(constants, links, header) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref uint GetLeftReference(uint node) => ref Links[node].LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref uint GetRightReference(uint node) => ref
        ↪  Links[node].RightAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetLeft(uint node) => Links[node].LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
21          protected override uint GetRight(uint node) => Links[node].RightAsSource;
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          protected override void SetLeft(uint node, uint left) => Links[node].LeftAsSource = left;
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          protected override void SetRight(uint node, uint right) => Links[node].RightAsSource =
            ↪  right;
28
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          protected override uint GetSize(uint node) => Links[node].SizeAsSource;
31
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          protected override void SetSize(uint node, uint size) => Links[node].SizeAsSource = size;
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          protected override uint GetTreeRoot() => Header->RootAsSource;
37
38          [MethodImpl(MethodImplOptions.AggressiveInlining)]
39          protected override uint GetBasePartValue(uint link) => Links[link].Source;
40
41          [MethodImpl(MethodImplOptions.AggressiveInlining)]
42          protected override bool FirstIsToTheLeftOfSecond(uint firstSource, uint firstTarget,
            ↪  uint secondSource, uint secondTarget)
43              => firstSource < secondSource || (firstSource == secondSource && firstTarget <
                ↪  secondTarget);
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected override bool FirstIsToTheRightOfSecond(uint firstSource, uint firstTarget,
            ↪  uint secondSource, uint secondTarget)
47              => firstSource > secondSource || (firstSource == secondSource && firstTarget >
                ↪  secondTarget);
48
49          [MethodImpl(MethodImplOptions.AggressiveInlining)]
50          protected override void ClearNode(uint node)
51          {
52              ref var link = ref Links[node];
53              link.LeftAsSource = 0U;
54              link.RightAsSource = 0U;
55              link.SizeAsSource = 0U;
56          }
57      }
58  }
```

1.94 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksTargetsRecursionlessSizeBalancedTre

```
1   using System.Runtime.CompilerServices;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Data.Doublets.Memory.United.Specific
6   {
7       public unsafe class UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods :
        ↪  UInt32LinksRecursionlessSizeBalancedTreeMethodsBase
8       {
9           public UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<uint>
            ↪  constants, RawLink<uint>* links, LinksHeader<uint>* header) : base(constants, links,
            ↪  header) { }
10
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          protected override ref uint GetLeftReference(uint node) => ref Links[node].LeftAsTarget;
13
14          [MethodImpl(MethodImplOptions.AggressiveInlining)]
15          protected override ref uint GetRightReference(uint node) => ref
            ↪  Links[node].RightAsTarget;
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          protected override uint GetLeft(uint node) => Links[node].LeftAsTarget;
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          protected override uint GetRight(uint node) => Links[node].RightAsTarget;
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          protected override void SetLeft(uint node, uint left) => Links[node].LeftAsTarget = left;
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          protected override void SetRight(uint node, uint right) => Links[node].RightAsTarget =
            ↪  right;
28
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        protected override uint GetSize(uint node) => Links[node].SizeAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(uint node, uint size) => Links[node].SizeAsTarget = size;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetTreeRoot() => Header->RootAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetBasePartValue(uint link) => Links[link].Target;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(uint firstSource, uint firstTarget,
        ↪  uint secondSource, uint secondTarget)
            => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
            ↪  secondSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(uint firstSource, uint firstTarget,
        ↪  uint secondSource, uint secondTarget)
            => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
            ↪  secondSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(uint node)
        {
            ref var link = ref Links[node];
            link.LeftAsTarget = 0U;
            link.RightAsTarget = 0U;
            link.SizeAsTarget = 0U;
        }
    }
}
```

1.95 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksTargetsSizeBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Specific
{
    public unsafe class UInt32LinksTargetsSizeBalancedTreeMethods :
    ↪  UInt32LinksSizeBalancedTreeMethodsBase
    {
        public UInt32LinksTargetsSizeBalancedTreeMethods(LinksConstants<uint> constants,
        ↪  RawLink<uint>* links, LinksHeader<uint>* header) : base(constants, links, header) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref uint GetLeftReference(uint node) => ref Links[node].LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref uint GetRightReference(uint node) => ref
        ↪  Links[node].RightAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetLeft(uint node) => Links[node].LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetRight(uint node) => Links[node].RightAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(uint node, uint left) => Links[node].LeftAsTarget = left;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(uint node, uint right) => Links[node].RightAsTarget =
        ↪  right;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetSize(uint node) => Links[node].SizeAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(uint node, uint size) => Links[node].SizeAsTarget = size;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetTreeRoot() => Header->RootAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override uint GetBasePartValue(uint link) => Links[link].Target;

```

```
41          [MethodImpl(MethodImplOptions.AggressiveInlining)]
42          protected override bool FirstIsToTheLeftOfSecond(uint firstSource, uint firstTarget,
            ↪  uint secondSource, uint secondTarget)
43              => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
                ↪  secondSource);
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected override bool FirstIsToTheRightOfSecond(uint firstSource, uint firstTarget,
            ↪  uint secondSource, uint secondTarget)
47              => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
                ↪  secondSource);
48
49          [MethodImpl(MethodImplOptions.AggressiveInlining)]
50          protected override void ClearNode(uint node)
51          {
52              ref var link = ref Links[node];
53              link.LeftAsTarget = 0U;
54              link.RightAsTarget = 0U;
55              link.SizeAsTarget = 0U;
56          }
57      }
58  }
```

1.96 ./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          [MethodImpl(MethodImplOptions.AggressiveInlining)]
23          public UInt32UnitedMemoryLinks(string address) : this(address, DefaultLinksSizeStep) { }
24
25          /// <summary>
26          /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
            ↪  минимальным шагом расширения базы данных.
27          /// </summary>
28          /// <param name="address">Полный пусть к файлу базы данных.</param>
29          /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
            ↪  байтах.</param>
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public UInt32UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
            ↪  FileMappedResizableDirectMemory(address, memoryReservationStep),
            ↪  memoryReservationStep) { }
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          public UInt32UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
            ↪  DefaultLinksSizeStep) { }
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          public UInt32UnitedMemoryLinks(IResizableDirectMemory memory, long
            ↪  memoryReservationStep) : this(memory, memoryReservationStep,
            ↪  Default<LinksConstants<uint>>.Instance, IndexTreeType.Default) { }
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          public UInt32UnitedMemoryLinks(IResizableDirectMemory memory, long
            ↪  memoryReservationStep, LinksConstants<uint> constants, IndexTreeType indexTreeType)
            ↪  : base(memory, memoryReservationStep, constants)
41          {
42              if (indexTreeType == IndexTreeType.SizeBalancedTree)
```

```
43              {
44                  _createSourceTreeMethods = () => new
                    ↪  UInt32LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
45                  _createTargetTreeMethods = () => new
                    ↪  UInt32LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
46              }
47              else
48              {
49                  _createSourceTreeMethods = () => new
                    ↪  UInt32LinksSourcesRecursionlessSizeBalancedTreeMethods(Constants, _links,
                    ↪  _header);
50                  _createTargetTreeMethods = () => new
                    ↪  UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods(Constants, _links,
                    ↪  _header);
51              }
52              Init(memory, memoryReservationStep);
53          }
54
55          [MethodImpl(MethodImplOptions.AggressiveInlining)]
56          protected override void SetPointers(IResizableDirectMemory memory)
57          {
58              _header = (LinksHeader<uint>*)memory.Pointer;
59              _links = (RawLink<uint>*)memory.Pointer;
60              SourcesTreeMethods = _createSourceTreeMethods();
61              TargetsTreeMethods = _createTargetTreeMethods();
62              UnusedLinksListMethods = new UInt32UnusedLinksListMethods(_links, _header);
63          }
64
65          [MethodImpl(MethodImplOptions.AggressiveInlining)]
66          protected override void ResetPointers()
67          {
68              base.ResetPointers();
69              _links = null;
70              _header = null;
71          }
72
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          protected override ref LinksHeader<uint> GetHeaderReference() => ref *_header;
75
76          [MethodImpl(MethodImplOptions.AggressiveInlining)]
77          protected override ref RawLink<uint> GetLinkReference(uint linkIndex) => ref
            ↪  _links[linkIndex];
78
79          [MethodImpl(MethodImplOptions.AggressiveInlining)]
80          protected override bool AreEqual(uint first, uint second) => first == second;
81
82          [MethodImpl(MethodImplOptions.AggressiveInlining)]
83          protected override bool LessThan(uint first, uint second) => first < second;
84
85          [MethodImpl(MethodImplOptions.AggressiveInlining)]
86          protected override bool LessOrEqualThan(uint first, uint second) => first <= second;
87
88          [MethodImpl(MethodImplOptions.AggressiveInlining)]
89          protected override bool GreaterThan(uint first, uint second) => first > second;
90
91          [MethodImpl(MethodImplOptions.AggressiveInlining)]
92          protected override bool GreaterOrEqualThan(uint first, uint second) => first >= second;
93
94          [MethodImpl(MethodImplOptions.AggressiveInlining)]
95          protected override uint GetZero() => 0U;
96
97          [MethodImpl(MethodImplOptions.AggressiveInlining)]
98          protected override uint GetOne() => 1U;
99
100         [MethodImpl(MethodImplOptions.AggressiveInlining)]
101         protected override long ConvertToInt64(uint value) => (long)value;
102
103         [MethodImpl(MethodImplOptions.AggressiveInlining)]
104         protected override uint ConvertToAddress(long value) => (uint)value;
105
106         [MethodImpl(MethodImplOptions.AggressiveInlining)]
107         protected override uint Add(uint first, uint second) => first + second;
108
109         [MethodImpl(MethodImplOptions.AggressiveInlining)]
110         protected override uint Subtract(uint first, uint second) => first - second;
111
112         [MethodImpl(MethodImplOptions.AggressiveInlining)]
113         protected override uint Increment(uint link) => ++link;
114
```

```
115         [MethodImpl(MethodImplOptions.AggressiveInlining)]
116         protected override uint Decrement(uint link) => --link;
117     }
118 }
```

1.97 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32UnusedLinksListMethods.cs

```
1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.Memory.United.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.United.Specific
7  {
8      public unsafe class UInt32UnusedLinksListMethods : UnusedLinksListMethods<uint>
9      {
10         private readonly RawLink<uint>* _links;
11         private readonly LinksHeader<uint>* _header;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public UInt32UnusedLinksListMethods(RawLink<uint>* links, LinksHeader<uint>* header)
15             : base((byte*)links, (byte*)header)
16         {
17             _links = links;
18             _header = header;
19         }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected override ref RawLink<uint> GetLinkReference(uint link) => ref _links[link];
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override ref LinksHeader<uint> GetHeaderReference() => ref *_header;
26     }
27 }
```

1.98 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksAvlBalancedTreeMethodsBase.cs

```
1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.Memory.United.Generic;
3  using static System.Runtime.CompilerServices.Unsafe;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Memory.United.Specific
8  {
9      public unsafe abstract class UInt64LinksAvlBalancedTreeMethodsBase :
       ↪  LinksAvlBalancedTreeMethodsBase<ulong>
10     {
11         protected new readonly RawLink<ulong>* Links;
12         protected new readonly LinksHeader<ulong>* Header;
13
14         protected UInt64LinksAvlBalancedTreeMethodsBase(LinksConstants<ulong> constants,
           ↪  RawLink<ulong>* links, LinksHeader<ulong>* header)
15             : base(constants, (byte*)links, (byte*)header)
16         {
17             Links = links;
18             Header = header;
19         }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected override ulong GetZero() => 0UL;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override bool EqualToZero(ulong value) => value == 0UL;
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override bool AreEqual(ulong first, ulong second) => first == second;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected override bool GreaterThanZero(ulong value) => value > 0UL;
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected override bool GreaterThan(ulong first, ulong second) => first > second;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
           ↪  always true for ulong
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is always >= 0 for ulong

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessThanZero(ulong value) => false; // value < 0 is always false for ulong

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessThan(ulong first, ulong second) => first < second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Increment(ulong value) => ++value;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Decrement(ulong value) => --value;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Add(ulong first, ulong second) => first + second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Subtract(ulong first, ulong second) => first - second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
        {
            ref var firstLink = ref Links[first];
            ref var secondLink = ref Links[second];
            return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target, secondLink.Source, secondLink.Target);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
        {
            ref var firstLink = ref Links[first];
            ref var secondLink = ref Links[second];
            return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target, secondLink.Source, secondLink.Target);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetSizeValue(ulong value) => (value & 4294967264UL) >> 5;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSizeValue(ref ulong storedValue, ulong size) => storedValue = storedValue & 31UL | (size & 134217727UL) << 5;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetLeftIsChildValue(ulong value) => (value & 16UL) >> 4 == 1UL;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeftIsChildValue(ref ulong storedValue, bool value) => storedValue = storedValue & 4294967279UL | (As<bool, byte>(ref value) & 1UL) << 4;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetRightIsChildValue(ulong value) => (value & 8UL) >> 3 == 1UL;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRightIsChildValue(ref ulong storedValue, bool value) => storedValue = storedValue & 4294967287UL | (As<bool, byte>(ref value) & 1UL) << 3;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override sbyte GetBalanceValue(ulong value) => unchecked((sbyte)(value & 7UL | 0xF8UL * ((value & 4UL) >> 2))); // if negative, then continue ones to the end of sbyte

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetBalanceValue(ref ulong storedValue, sbyte value) => storedValue = unchecked(storedValue & 4294967288UL | (ulong)((byte)value >> 5 & 4 | value & 3) & 7UL);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
```

```
    }
}
```

1.99 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksRecursionlessSizeBalancedTreeMetho

```
using System.Runtime.CompilerServices;
using Platform.Data.Doublets.Memory.United.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Specific
{
    public unsafe abstract class UInt64LinksRecursionlessSizeBalancedTreeMethodsBase :
    ↪ LinksRecursionlessSizeBalancedTreeMethodsBase<ulong>
    {
        protected new readonly RawLink<ulong>* Links;
        protected new readonly LinksHeader<ulong>* Header;

        protected UInt64LinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<ulong>
        ↪ constants, RawLink<ulong>* links, LinksHeader<ulong>* header)
            : base(constants, (byte*)links, (byte*)header)
        {
            Links = links;
            Header = header;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetZero() => 0UL;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool EqualToZero(ulong value) => value == 0UL;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool AreEqual(ulong first, ulong second) => first == second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThanZero(ulong value) => value > 0UL;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThan(ulong first, ulong second) => first > second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
        ↪ always true for ulong

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
        ↪ always >= 0 for ulong

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
        ↪ for ulong

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessThan(ulong first, ulong second) => first < second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Increment(ulong value) => ++value;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Decrement(ulong value) => --value;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Add(ulong first, ulong second) => first + second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Subtract(ulong first, ulong second) => first - second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
        {
            ref var firstLink = ref Links[first];
            ref var secondLink = ref Links[second];
            return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
            ↪ secondLink.Source, secondLink.Target);
```

```
71          }
72
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
75          {
76              ref var firstLink = ref Links[first];
77              ref var secondLink = ref Links[second];
78              return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
79          }
80
81          [MethodImpl(MethodImplOptions.AggressiveInlining)]
82          protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
83
84          [MethodImpl(MethodImplOptions.AggressiveInlining)]
85          protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
86      }
87  }
```

1.100 ./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       public unsafe abstract class UInt64LinksSizeBalancedTreeMethodsBase :
        ↪  LinksSizeBalancedTreeMethodsBase<ulong>
9       {
10          protected new readonly RawLink<ulong>* Links;
11          protected new readonly LinksHeader<ulong>* Header;
12
13          protected UInt64LinksSizeBalancedTreeMethodsBase(LinksConstants<ulong> constants,
            ↪  RawLink<ulong>* links, LinksHeader<ulong>* header)
14              : base(constants, (byte*)links, (byte*)header)
15          {
16              Links = links;
17              Header = header;
18          }
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          protected override ulong GetZero() => 0UL;
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          protected override bool EqualToZero(ulong value) => value == 0UL;
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          protected override bool AreEqual(ulong first, ulong second) => first == second;
28
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          protected override bool GreaterThanZero(ulong value) => value > 0UL;
31
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          protected override bool GreaterThan(ulong first, ulong second) => first > second;
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
37
38          [MethodImpl(MethodImplOptions.AggressiveInlining)]
39          protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
            ↪  always true for ulong
40
41          [MethodImpl(MethodImplOptions.AggressiveInlining)]
42          protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
            ↪  always >= 0 for ulong
43
44          [MethodImpl(MethodImplOptions.AggressiveInlining)]
45          protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
46
47          [MethodImpl(MethodImplOptions.AggressiveInlining)]
48          protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
            ↪  for ulong
49
50          [MethodImpl(MethodImplOptions.AggressiveInlining)]
51          protected override bool LessThan(ulong first, ulong second) => first < second;
52
53          [MethodImpl(MethodImplOptions.AggressiveInlining)]
54          protected override ulong Increment(ulong value) => ++value;
55
```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Decrement(ulong value) => --value;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Add(ulong first, ulong second) => first + second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Subtract(ulong first, ulong second) => first - second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
        {
            ref var firstLink = ref Links[first];
            ref var secondLink = ref Links[second];
            return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
            ↪  secondLink.Source, secondLink.Target);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
        {
            ref var firstLink = ref Links[first];
            ref var secondLink = ref Links[second];
            return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
            ↪  secondLink.Source, secondLink.Target);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
    }
}
```

1.101 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesAvlBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Specific
{
    public unsafe class UInt64LinksSourcesAvlBalancedTreeMethods :
    ↪  UInt64LinksAvlBalancedTreeMethodsBase
    {
        public UInt64LinksSourcesAvlBalancedTreeMethods(LinksConstants<ulong> constants,
        ↪  RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
        ↪  { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetLeftReference(ulong node) => ref
        ↪  Links[node].LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetRightReference(ulong node) => ref
        ↪  Links[node].RightAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetRight(ulong node) => Links[node].RightAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
        ↪  left;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
        ↪  right;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetSize(ulong node) => GetSizeValue(Links[node].SizeAsSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
        ↪  Links[node].SizeAsSource, size);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
36          protected override bool GetLeftIsChild(ulong node) =>
            ↪  GetLeftIsChildValue(Links[node].SizeAsSource);
37
38          //[MethodImpl(MethodImplOptions.AggressiveInlining)]
39          //protected override bool GetLeftIsChild(ulong node) => IsChild(node, GetLeft(node));
40
41          [MethodImpl(MethodImplOptions.AggressiveInlining)]
42          protected override void SetLeftIsChild(ulong node, bool value) =>
            ↪  SetLeftIsChildValue(ref Links[node].SizeAsSource, value);
43
44          [MethodImpl(MethodImplOptions.AggressiveInlining)]
45          protected override bool GetRightIsChild(ulong node) =>
            ↪  GetRightIsChildValue(Links[node].SizeAsSource);
46
47          //[MethodImpl(MethodImplOptions.AggressiveInlining)]
48          //protected override bool GetRightIsChild(ulong node) => IsChild(node, GetRight(node));
49
50          [MethodImpl(MethodImplOptions.AggressiveInlining)]
51          protected override void SetRightIsChild(ulong node, bool value) =>
            ↪  SetRightIsChildValue(ref Links[node].SizeAsSource, value);
52
53          [MethodImpl(MethodImplOptions.AggressiveInlining)]
54          protected override sbyte GetBalance(ulong node) =>
            ↪  GetBalanceValue(Links[node].SizeAsSource);
55
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
            ↪  Links[node].SizeAsSource, value);
58
59          [MethodImpl(MethodImplOptions.AggressiveInlining)]
60          protected override ulong GetTreeRoot() => Header->RootAsSource;
61
62          [MethodImpl(MethodImplOptions.AggressiveInlining)]
63          protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
64
65          [MethodImpl(MethodImplOptions.AggressiveInlining)]
66          protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
            ↪  ulong secondSource, ulong secondTarget)
67              => firstSource < secondSource || (firstSource == secondSource && firstTarget <
                ↪  secondTarget);
68
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
            ↪  ulong secondSource, ulong secondTarget)
71              => firstSource > secondSource || (firstSource == secondSource && firstTarget >
                ↪  secondTarget);
72
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          protected override void ClearNode(ulong node)
75          {
76              ref var link = ref Links[node];
77              link.LeftAsSource = OUL;
78              link.RightAsSource = OUL;
79              link.SizeAsSource = OUL;
80          }
81      }
82  }
```

1.102 ./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       public unsafe class UInt64LinksSourcesRecursionlessSizeBalancedTreeMethods :
        ↪  UInt64LinksRecursionlessSizeBalancedTreeMethodsBase
8       {
9           public UInt64LinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<ulong>
            ↪  constants, RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants,
            ↪  links, header) { }
10
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          protected override ref ulong GetLeftReference(ulong node) => ref
            ↪  Links[node].LeftAsSource;
13
14          [MethodImpl(MethodImplOptions.AggressiveInlining)]
15          protected override ref ulong GetRightReference(ulong node) => ref
            ↪  Links[node].RightAsSource;
```

```

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetRight(ulong node) => Links[node].RightAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
        ↪  left;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
        ↪  right;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetSize(ulong node) => Links[node].SizeAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsSource =
        ↪  size;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetTreeRoot() => Header->RootAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetBasePartValue(ulong link) => Links[link].Source;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
        ↪  ulong secondSource, ulong secondTarget)
            => firstSource < secondSource || (firstSource == secondSource && firstTarget <
            ↪  secondTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
        ↪  ulong secondSource, ulong secondTarget)
            => firstSource > secondSource || (firstSource == secondSource && firstTarget >
            ↪  secondTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(ulong node)
        {
            ref var link = ref Links[node];
            link.LeftAsSource = 0UL;
            link.RightAsSource = 0UL;
            link.SizeAsSource = 0UL;
        }
    }
}
```

1.103 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesSizeBalancedTreeMethods.c

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Specific
{
    public unsafe class UInt64LinksSourcesSizeBalancedTreeMethods :
    ↪  UInt64LinksSizeBalancedTreeMethodsBase
    {
        public UInt64LinksSourcesSizeBalancedTreeMethods(LinksConstants<ulong> constants,
        ↪  RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
        ↪  { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetLeftReference(ulong node) => ref
        ↪  Links[node].LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetRightReference(ulong node) => ref
        ↪  Links[node].RightAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetRight(ulong node) => Links[node].RightAsSource;

```

```
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
            ↪  left;
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
            ↪  right;
28
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          protected override ulong GetSize(ulong node) => Links[node].SizeAsSource;
31
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsSource =
            ↪  size;
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          protected override ulong GetTreeRoot() => Header->RootAsSource;
37
38          [MethodImpl(MethodImplOptions.AggressiveInlining)]
39          protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
40
41          [MethodImpl(MethodImplOptions.AggressiveInlining)]
42          protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
            ↪  ulong secondSource, ulong secondTarget)
43              => firstSource < secondSource || (firstSource == secondSource && firstTarget <
                ↪  secondTarget);
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
            ↪  ulong secondSource, ulong secondTarget)
47              => firstSource > secondSource || (firstSource == secondSource && firstTarget >
                ↪  secondTarget);
48
49          [MethodImpl(MethodImplOptions.AggressiveInlining)]
50          protected override void ClearNode(ulong node)
51          {
52              ref var link = ref Links[node];
53              link.LeftAsSource = OUL;
54              link.RightAsSource = OUL;
55              link.SizeAsSource = OUL;
56          }
57      }
58  }
```

1.104 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsAvlBalancedTreeMethods.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       public unsafe class UInt64LinksTargetsAvlBalancedTreeMethods :
        ↪  UInt64LinksAvlBalancedTreeMethodsBase
8       {
9           public UInt64LinksTargetsAvlBalancedTreeMethods(LinksConstants<ulong> constants,
            ↪  RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
            ↪  { }
10
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          protected override ref ulong GetLeftReference(ulong node) => ref
            ↪  Links[node].LeftAsTarget;
13
14          [MethodImpl(MethodImplOptions.AggressiveInlining)]
15          protected override ref ulong GetRightReference(ulong node) => ref
            ↪  Links[node].RightAsTarget;
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
            ↪  left;
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
            ↪  right;
```

```

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetSize(ulong node) => GetSizeValue(Links[node].SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
        ↪  Links[node].SizeAsTarget, size);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetLeftIsChild(ulong node) =>
        ↪  GetLeftIsChildValue(Links[node].SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeftIsChild(ulong node, bool value) =>
        ↪  SetLeftIsChildValue(ref Links[node].SizeAsTarget, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetRightIsChild(ulong node) =>
        ↪  GetRightIsChildValue(Links[node].SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRightIsChild(ulong node, bool value) =>
        ↪  SetRightIsChildValue(ref Links[node].SizeAsTarget, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override sbyte GetBalance(ulong node) =>
        ↪  GetBalanceValue(Links[node].SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
        ↪  Links[node].SizeAsTarget, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetTreeRoot() => Header->RootAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetBasePartValue(ulong link) => Links[link].Target;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
        ↪  ulong secondSource, ulong secondTarget)
            => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
            ↪  secondSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
        ↪  ulong secondSource, ulong secondTarget)
            => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
            ↪  secondSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(ulong node)
        {
            ref var link = ref Links[node];
            link.LeftAsTarget = OUL;
            link.RightAsTarget = OUL;
            link.SizeAsTarget = OUL;
        }
    }
}
```

1.105 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsRecursionlessSizeBalancedTr

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Specific
{
    public unsafe class UInt64LinksTargetsRecursionlessSizeBalancedTreeMethods :
    ↪  UInt64LinksRecursionlessSizeBalancedTreeMethodsBase
    {
        public UInt64LinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<ulong>
        ↪  constants, RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants,
        ↪  links, header) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetLeftReference(ulong node) => ref
        ↪  Links[node].LeftAsTarget;
```

```

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetRightReference(ulong node) => ref
        ↪  Links[node].RightAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
        ↪  left;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
        ↪  right;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsTarget =
        ↪  size;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetTreeRoot() => Header->RootAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetBasePartValue(ulong link) => Links[link].Target;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
        ↪  ulong secondSource, ulong secondTarget)
            => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
            ↪  secondSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
        ↪  ulong secondSource, ulong secondTarget)
            => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
            ↪  secondSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(ulong node)
        {
            ref var link = ref Links[node];
            link.LeftAsTarget = 0UL;
            link.RightAsTarget = 0UL;
            link.SizeAsTarget = 0UL;
        }
    }
}
```

1.106 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsSizeBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Specific
{
    public unsafe class UInt64LinksTargetsSizeBalancedTreeMethods :
    ↪  UInt64LinksSizeBalancedTreeMethodsBase
    {
        public UInt64LinksTargetsSizeBalancedTreeMethods(LinksConstants<ulong> constants,
        ↪  RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
        ↪  { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetLeftReference(ulong node) => ref
        ↪  Links[node].LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetRightReference(ulong node) => ref
        ↪  Links[node].RightAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;
```

```

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
        ↪  left;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
        ↪  right;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsTarget =
        ↪  size;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetTreeRoot() => Header->RootAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetBasePartValue(ulong link) => Links[link].Target;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
        ↪  ulong secondSource, ulong secondTarget)
            => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
            ↪  secondSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
        ↪  ulong secondSource, ulong secondTarget)
            => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
            ↪  secondSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(ulong node)
        {
            ref var link = ref Links[node];
            link.LeftAsTarget = 0UL;
            link.RightAsTarget = 0UL;
            link.SizeAsTarget = 0UL;
        }
    }
}
```

1.107 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnitedMemoryLinks.cs

```
using System;
using System.Runtime.CompilerServices;
using Platform.Memory;
using Platform.Singletons;
using Platform.Data.Doublets.Memory.United.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Memory.United.Specific
{
    /// <summary>
    /// <para>Represents a low-level implementation of direct access to resizable memory, for
    ↪  organizing the storage of links with addresses represented as <see cref="ulong"
    ↪  />.</para>
    /// <para>Представляет низкоуровневую реализация прямого доступа к памяти с переменным
    ↪  размером, для организации хранения связей с адресами представленными в виде <see
    ↪  cref="ulong"/>.</para>
    /// </summary>
    public unsafe class UInt64UnitedMemoryLinks : UnitedMemoryLinksBase<ulong>
    {
        private readonly Func<ILinksTreeMethods<ulong>> _createSourceTreeMethods;
        private readonly Func<ILinksTreeMethods<ulong>> _createTargetTreeMethods;
        private LinksHeader<ulong>* _header;
        private RawLink<ulong>* _links;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64UnitedMemoryLinks(string address) : this(address, DefaultLinksSizeStep) { }

        /// <summary>
        /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
        ↪  минимальным шагом расширения базы данных.
```

```
27          /// </summary>
28          /// <param name="address">Полный пусть к файлу базы данных.</param>
29          /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
            ↪  байтах.</param>
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public UInt64UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
            ↪  FileMappedResizableDirectMemory(address, memoryReservationStep),
            ↪  memoryReservationStep) { }
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          public UInt64UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
            ↪  DefaultLinksSizeStep) { }
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          public UInt64UnitedMemoryLinks(IResizableDirectMemory memory, long
            ↪  memoryReservationStep) : this(memory, memoryReservationStep,
            ↪  Default<LinksConstants<ulong>>.Instance, IndexTreeType.Default) { }
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          public UInt64UnitedMemoryLinks(IResizableDirectMemory memory, long
            ↪  memoryReservationStep, LinksConstants<ulong> constants, IndexTreeType indexTreeType)
            ↪  : base(memory, memoryReservationStep, constants)
41          {
42              if (indexTreeType == IndexTreeType.SizedAndThreadedAVLBalancedTree)
43              {
44                  _createSourceTreeMethods = () => new
                    ↪  UInt64LinksSourcesAvlBalancedTreeMethods(Constants, _links, _header);
45                  _createTargetTreeMethods = () => new
                    ↪  UInt64LinksTargetsAvlBalancedTreeMethods(Constants, _links, _header);
46              }
47              else if (indexTreeType == IndexTreeType.SizeBalancedTree)
48              {
49                  _createSourceTreeMethods = () => new
                    ↪  UInt64LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
50                  _createTargetTreeMethods = () => new
                    ↪  UInt64LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
51              }
52              else
53              {
54                  _createSourceTreeMethods = () => new
                    ↪  UInt64LinksSourcesRecursionlessSizeBalancedTreeMethods(Constants, _links,
                    ↪  _header);
55                  _createTargetTreeMethods = () => new
                    ↪  UInt64LinksTargetsRecursionlessSizeBalancedTreeMethods(Constants, _links,
                    ↪  _header);
56              }
57              Init(memory, memoryReservationStep);
58          }
59
60          [MethodImpl(MethodImplOptions.AggressiveInlining)]
61          protected override void SetPointers(IResizableDirectMemory memory)
62          {
63              _header = (LinksHeader<ulong>*)memory.Pointer;
64              _links = (RawLink<ulong>*)memory.Pointer;
65              SourcesTreeMethods = _createSourceTreeMethods();
66              TargetsTreeMethods = _createTargetTreeMethods();
67              UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_links, _header);
68          }
69
70          [MethodImpl(MethodImplOptions.AggressiveInlining)]
71          protected override void ResetPointers()
72          {
73              base.ResetPointers();
74              _links = null;
75              _header = null;
76          }
77
78          [MethodImpl(MethodImplOptions.AggressiveInlining)]
79          protected override ref LinksHeader<ulong> GetHeaderReference() => ref *_header;
80
81          [MethodImpl(MethodImplOptions.AggressiveInlining)]
82          protected override ref RawLink<ulong> GetLinkReference(ulong linkIndex) => ref
            ↪  _links[linkIndex];
83
84          [MethodImpl(MethodImplOptions.AggressiveInlining)]
85          protected override bool AreEqual(ulong first, ulong second) => first == second;
86
```

```
87          [MethodImpl(MethodImplOptions.AggressiveInlining)]
88          protected override bool LessThan(ulong first, ulong second) => first < second;
89
90          [MethodImpl(MethodImplOptions.AggressiveInlining)]
91          protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
92
93          [MethodImpl(MethodImplOptions.AggressiveInlining)]
94          protected override bool GreaterThan(ulong first, ulong second) => first > second;
95
96          [MethodImpl(MethodImplOptions.AggressiveInlining)]
97          protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
98
99          [MethodImpl(MethodImplOptions.AggressiveInlining)]
100         protected override ulong GetZero() => 0UL;
101
102         [MethodImpl(MethodImplOptions.AggressiveInlining)]
103         protected override ulong GetOne() => 1UL;
104
105         [MethodImpl(MethodImplOptions.AggressiveInlining)]
106         protected override long ConvertToInt64(ulong value) => (long)value;
107
108         [MethodImpl(MethodImplOptions.AggressiveInlining)]
109         protected override ulong ConvertToAddress(long value) => (ulong)value;
110
111         [MethodImpl(MethodImplOptions.AggressiveInlining)]
112         protected override ulong Add(ulong first, ulong second) => first + second;
113
114         [MethodImpl(MethodImplOptions.AggressiveInlining)]
115         protected override ulong Subtract(ulong first, ulong second) => first - second;
116
117         [MethodImpl(MethodImplOptions.AggressiveInlining)]
118         protected override ulong Increment(ulong link) => ++link;
119
120         [MethodImpl(MethodImplOptions.AggressiveInlining)]
121         protected override ulong Decrement(ulong link) => --link;
122     }
123 }
```

1.108 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnusedLinksListMethods.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      public unsafe class UInt64UnusedLinksListMethods : UnusedLinksListMethods<ulong>
9      {
10         private readonly RawLink<ulong>* _links;
11         private readonly LinksHeader<ulong>* _header;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public UInt64UnusedLinksListMethods(RawLink<ulong>* links, LinksHeader<ulong>* header)
15             : base((byte*)links, (byte*)header)
16         {
17             _links = links;
18             _header = header;
19         }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref _links[link];
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override ref LinksHeader<ulong> GetHeaderReference() => ref *_header;
26     }
27 }
```

1.109 ./csharp/Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Interfaces;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.PropertyOperators
8  {
9      public class PropertiesOperator<TLink> : LinksOperatorBase<TLink>, IProperties<TLink, TLink,
       ↪  TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
```

```

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public PropertiesOperator(ILinks<TLink> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink GetValue(TLink @object, TLink property)
        {
            var links = _links;
            var objectProperty = links.SearchOrDefault(@object, property);
            if (_equalityComparer.Equals(objectProperty, default))
            {
                return default;
            }
            var constants = links.Constants;
            var any = constants.Any;
            var query = new Link<TLink>(any, objectProperty, any);
            var valueLink = links.SingleOrDefault(query);
            if (valueLink == null)
            {
                return default;
            }
            return links.GetTarget(valueLink[constants.IndexPart]);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void SetValue(TLink @object, TLink property, TLink value)
        {
            var links = _links;
            var objectProperty = links.GetOrCreate(@object, property);
            links.DeleteMany(links.AllIndices(links.Constants.Any, objectProperty));
            links.GetOrCreate(objectProperty, value);
        }
    }
}
```

1.110 ./csharp/Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Interfaces;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.PropertyOperators
{
    public class PropertyOperator<TLink> : LinksOperatorBase<TLink>, IProperty<TLink, TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly TLink _propertyMarker;
        private readonly TLink _propertyValueMarker;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public PropertyOperator(ILinks<TLink> links, TLink propertyMarker, TLink
        ↪  propertyValueMarker) : base(links)
        {
            _propertyMarker = propertyMarker;
            _propertyValueMarker = propertyValueMarker;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Get(TLink link)
        {
            var property = _links.SearchOrDefault(link, _propertyMarker);
            return GetValue(GetContainer(property));
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private TLink GetContainer(TLink property)
        {
            var valueContainer = default(TLink);
            if (_equalityComparer.Equals(property, default))
            {
                return valueContainer;
            }
            var links = _links;
            var constants = links.Constants;
            var countinueConstant = constants.Continue;
            var breakConstant = constants.Break;
            var anyConstant = constants.Any;
```

```
            var query = new Link<TLink>(anyConstant, property, anyConstant);
            links.Each(candidate =>
            {
                var candidateTarget = links.GetTarget(candidate);
                var valueTarget = links.GetTarget(candidateTarget);
                if (_equalityComparer.Equals(valueTarget, _propertyValueMarker))
                {
                    valueContainer = links.GetIndex(candidate);
                    return breakConstant;
                }
                return countinueConstant;
            }, query);
            return valueContainer;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private TLink GetValue(TLink container) => _equalityComparer.Equals(container, default)
        ↪  ? default : _links.GetTarget(container);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Set(TLink link, TLink value)
        {
            var links = _links;
            var property = links.GetOrCreate(link, _propertyMarker);
            var container = GetContainer(property);
            if (_equalityComparer.Equals(container, default))
            {
                links.GetOrCreate(property, value);
            }
            else
            {
                links.Update(container, property, value);
            }
        }
    }
}
```

1.111 ./csharp/Platform.Data.Doublets/Stacks/Stack.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections.Stacks;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Stacks
{
    public class Stack<TLink> : LinksOperatorBase<TLink>, IStack<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly TLink _stack;

        public bool IsEmpty
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _equalityComparer.Equals(Peek(), _stack);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Stack(ILinks<TLink> links, TLink stack) : base(links) => _stack = stack;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private TLink GetStackMarker() => _links.GetSource(_stack);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private TLink GetTop() => _links.GetTarget(_stack);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Peek() => _links.GetTarget(GetTop());

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Pop()
        {
            var element = Peek();
            if (!_equalityComparer.Equals(element, _stack))
            {
                var top = GetTop();
                var previousTop = _links.GetSource(top);
                _links.Update(_stack, GetStackMarker(), previousTop);
```

```
                _links.Delete(top);
            }
            return element;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
        ↪  _links.GetOrCreate(GetTop(), element));
    }
}
```

1.112 ./csharp/Platform.Data.Doublets/Stacks/StackExtensions.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Stacks
{
    public static class StackExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLink CreateStack<TLink>(this ILinks<TLink> links, TLink stackMarker)
        {
            var stackPoint = links.CreatePoint();
            var stack = links.Update(stackPoint, stackMarker, stackPoint);
            return stack;
        }
    }
}
```

1.113 ./csharp/Platform.Data.Doublets/SynchronizedLinks.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Data.Doublets;
using Platform.Threading.Synchronization;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets
{
    /// <remarks>
    /// TODO: Autogeneration of synchronized wrapper (decorator).
    /// TODO: Try to unfold code of each method using IL generation for performance improvements.
    /// TODO: Or even to unfold multiple layers of implementations.
    /// </remarks>
    public class SynchronizedLinks<TLinkAddress> : ISynchronizedLinks<TLinkAddress>
    {
        public LinksConstants<TLinkAddress> Constants
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        public ISynchronization SyncRoot
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        public ILinks<TLinkAddress> Sync
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        public ILinks<TLinkAddress> Unsync
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SynchronizedLinks(ILinks<TLinkAddress> links) : this(new
        ↪  ReaderWriterLockSynchronization(), links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SynchronizedLinks(ISynchronization synchronization, ILinks<TLinkAddress> links)
        {
            SyncRoot = synchronization;
```

```
            Sync = this;
            Unsync = links;
            Constants = links.Constants;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Count(IList<TLinkAddress> restriction) =>
        ↪ SyncRoot.ExecuteReadOperation(restriction, Unsync.Count);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler,
        ↪ IList<TLinkAddress> restrictions) => SyncRoot.ExecuteReadOperation(handler,
        ↪ restrictions, (handler1, restrictions1) => Unsync.Each(handler1, restrictions1));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Create(IList<TLinkAddress> restrictions) =>
        ↪ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Create);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress>
        ↪ substitution) => SyncRoot.ExecuteWriteOperation(restrictions, substitution,
        ↪ Unsync.Update);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Delete(IList<TLinkAddress> restrictions) =>
        ↪ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Delete);

        //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
        ↪ IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
        //{
        //    if (restriction != null && substitution != null &&
        ↪ !substitution.EqualTo(restriction))
        //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
        ↪ substitution, substitutedHandler, Unsync.Trigger);

        //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
        ↪ substitutedHandler, Unsync.Trigger);
        //}
    }
}
```

1.114 ./csharp/Platform.Data.Doublets/UInt64LinksExtensions.cs

```
using System;
using System.Text;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Singletons;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets
{
    public static class UInt64LinksExtensions
    {
        public static readonly LinksConstants<ulong> Constants =
        ↪ Default<LinksConstants<ulong>>.Instance;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
        {
            if (sequence == null)
            {
                return false;
            }
            var constants = links.Constants;
            for (var i = 0; i < sequence.Length; i++)
            {
                if (sequence[i] == constants.Any)
                {
                    return true;
                }
            }
            return false;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
        ↪ Func<Link<ulong>, bool> isElement, bool renderIndex = false, bool renderDebug =
        ↪ false)
```

```
35          {
36              var sb = new StringBuilder();
37              var visited = new HashSet<ulong>();
38              links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
                ↪  innerSb.Append(link.Index), renderIndex, renderDebug);
39              return sb.ToString();
40          }
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          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)
44          {
45              var sb = new StringBuilder();
46              var visited = new HashSet<ulong>();
47              links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
                ↪  renderDebug);
48              return sb.ToString();
49          }
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          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)
53          {
54              if (sb == null)
55              {
56                  throw new ArgumentNullException(nameof(sb));
57              }
58              if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
                ↪  Constants.Itself)
59              {
60                  return;
61              }
62              if (links.Exists(linkIndex))
63              {
64                  if (visited.Add(linkIndex))
65                  {
66                      sb.Append('(');
67                      var link = new Link<ulong>(links.GetLink(linkIndex));
68                      if (renderIndex)
69                      {
70                          sb.Append(link.Index);
71                          sb.Append(':');
72                      }
73                      if (link.Source == link.Index)
74                      {
75                          sb.Append(link.Index);
76                      }
77                      else
78                      {
79                          var source = new Link<ulong>(links.GetLink(link.Source));
80                          if (isElement(source))
81                          {
82                              appendElement(sb, source);
83                          }
84                          else
85                          {
86                              links.AppendStructure(sb, visited, source.Index, isElement,
                                ↪  appendElement, renderIndex);
87                          }
88                      }
89                      sb.Append(' ');
90                      if (link.Target == link.Index)
91                      {
92                          sb.Append(link.Index);
93                      }
94                      else
95                      {
96                          var target = new Link<ulong>(links.GetLink(link.Target));
97                          if (isElement(target))
98                          {
99                              appendElement(sb, target);
100                         }
101                         else
102                         {
```

```
                            links.AppendStructure(sb, visited, target.Index, isElement,
                            ↪  appendElement, renderIndex);
                        }
                    }
                    sb.Append(')');
                }
                else
                {
                    if (renderDebug)
                    {
                        sb.Append('*');
                    }
                    sb.Append(linkIndex);
                }
            }
            else
            {
                if (renderDebug)
                {
                    sb.Append('~');
                }
                sb.Append(linkIndex);
            }
        }
    }
}
```

1.115 ./csharp/Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs

```
using System;
using System.Linq;
using System.Collections.Generic;
using System.IO;
using System.Runtime.CompilerServices;
using System.Threading;
using System.Threading.Tasks;
using Platform.Disposables;
using Platform.Timestamps;
using Platform.Unsafe;
using Platform.IO;
using Platform.Data.Doublets.Decorators;
using Platform.Exceptions;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets
{
    public class UInt64LinksTransactionsLayer : LinksDisposableDecoratorBase<ulong> //-V3073
    {
        /// <remarks>
        /// Альтернативные варианты хранения трансформации (элемента транзакции):
        ///
        /// private enum TransitionType
        /// {
        ///     Creation,
        ///     UpdateOf,
        ///     UpdateTo,
        ///     Deletion
        /// }
        ///
        /// private struct Transition
        /// {
        ///     public ulong TransactionId;
        ///     public UniqueTimestamp Timestamp;
        ///     public TransactionItemType Type;
        ///     public Link Source;
        ///     public Link Linker;
        ///     public Link Target;
        /// }
        ///
        /// Или
        ///
        /// public struct TransitionHeader
        /// {
        ///     public ulong TransactionIdCombined;
        ///     public ulong TimestampCombined;
        ///
        ///     public ulong TransactionId
        ///     {
        ///         get
```

```
///         {
///             return (ulong) mask &amp;amp; TransactionIdCombined;
///         }
///     }
///
///     public UniqueTimestamp Timestamp
///     {
///         get
///         {
///             return (UniqueTimestamp)mask &amp;amp; TransactionIdCombined;
///         }
///     }
///
///     public TransactionItemType Type
///     {
///         get
///         {
///             // Использовать по одному биту из TransactionId и Timestamp,
///             // для значения в 2 бита, которое представляет тип операции
///             throw new NotImplementedException();
///         }
///     }
/// }
///
/// private struct Transition
/// {
///     public TransitionHeader Header;
///     public Link Source;
///     public Link Linker;
///     public Link Target;
/// }
///
/// </remarks>
public struct Transition : IEquatable<Transition>
{
    public static readonly long Size = Structure<Transition>.Size;

    public readonly ulong TransactionId;
    public readonly Link<ulong> Before;
    public readonly Link<ulong> After;
    public readonly Timestamp Timestamp;

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
    ↪  transactionId, Link<ulong> before, Link<ulong> after)
    {
        TransactionId = transactionId;
        Before = before;
        After = after;
        Timestamp = uniqueTimestampFactory.Create();
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
    ↪  transactionId, Link<ulong> before) : this(uniqueTimestampFactory, transactionId,
    ↪  before, default) { }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
    ↪  transactionId) : this(uniqueTimestampFactory, transactionId, default, default) {
    ↪  }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
    ↪  {After}";

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public override bool Equals(object obj) => obj is Transition transition ?
    ↪  Equals(transition) : false;

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public override int GetHashCode() => (TransactionId, Before, After,
    ↪  Timestamp).GetHashCode();

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public bool Equals(Transition other) => TransactionId == other.TransactionId &&
    ↪  Before == other.Before && After == other.After && Timestamp == other.Timestamp;

```

```
121             [MethodImpl(MethodImplOptions.AggressiveInlining)]
122             public static bool operator ==(Transition left, Transition right) =>
                ↪ left.Equals(right);
123
124             [MethodImpl(MethodImplOptions.AggressiveInlining)]
125             public static bool operator !=(Transition left, Transition right) => !(left ==
                ↪ right);
126         }
127
128         /// <remarks>
129         /// Другие варианты реализации транзакций (атомарности):
130         ///     1. Разделение хранения значения связи ((Source Target) или (Source Linker
            ↪ Target)) и индексов.
131         ///     2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
            ↪ потребуется решить вопрос
132         ///        со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
            ↪ пересечениями идентификаторов.
133         ///
134         /// Где хранить промежуточный список транзакций?
135         ///
136         /// В оперативной памяти:
137         ///  Минусы:
138         ///     1. Может усложнить систему, если она будет функционировать самостоятельно,
139         ///     так как нужно отдельно выделять память под список трансформаций.
140         ///     2. Выделенной оперативной памяти может не хватить, в том случае,
141         ///     если транзакция использует слишком много трансформаций.
142         ///         -> Можно использовать жёсткий диск для слишком длинных транзакций.
143         ///         -> Максимальный размер списка трансформаций можно ограничить / задать
            ↪ константой.
144         ///     3. При подтверждении транзакции (Commit) все трансформации записываются разом
            ↪ создавая задержку.
145         ///
146         /// На жёстком диске:
147         ///  Минусы:
148         ///     1. Длительный отклик, на запись каждой трансформации.
149         ///     2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
150         ///         -> Это может решаться упаковкой/исключением дублирующих операций.
151         ///         -> Также это может решаться тем, что короткие транзакции вообще
152         ///            не будут записываться в случае отката.
153         ///     3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
            ↪ операции (трансформации)
154         ///        будут записаны в лог.
155         ///
156         /// </remarks>
157         public class Transaction : DisposableBase
158         {
159             private readonly Queue<Transition> _transitions;
160             private readonly UInt64LinksTransactionsLayer _layer;
161             public bool IsCommitted { get; private set; }
162             public bool IsReverted { get; private set; }
163
164             [MethodImpl(MethodImplOptions.AggressiveInlining)]
165             public Transaction(UInt64LinksTransactionsLayer layer)
166             {
167                 _layer = layer;
168                 if (_layer._currentTransactionId != 0)
169                 {
170                     throw new NotSupportedException("Nested transactions not supported.");
171                 }
172                 IsCommitted = false;
173                 IsReverted = false;
174                 _transitions = new Queue<Transition>();
175                 SetCurrentTransaction(layer, this);
176             }
177
178             [MethodImpl(MethodImplOptions.AggressiveInlining)]
179             public void Commit()
180             {
181                 EnsureTransactionAllowsWriteOperations(this);
182                 while (_transitions.Count > 0)
183                 {
184                     var transition = _transitions.Dequeue();
185                     _layer._transitions.Enqueue(transition);
186                 }
187                 _layer._lastCommitedTransactionId = _layer._currentTransactionId;
188                 IsCommitted = true;
189             }
190
```

```
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private void Revert()
            {
                EnsureTransactionAllowsWriteOperations(this);
                var transitionsToRevert = new Transition[_transitions.Count];
                _transitions.CopyTo(transitionsToRevert, 0);
                for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
                {
                    _layer.RevertTransition(transitionsToRevert[i]);
                }
                IsReverted = true;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
            ↪  Transaction transaction)
            {
                layer._currentTransactionId = layer._lastCommitedTransactionId + 1;
                layer._currentTransactionTransitions = transaction._transitions;
                layer._currentTransaction = transaction;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
            {
                if (transaction.IsReverted)
                {
                    throw new InvalidOperationException("Transation is reverted.");
                }
                if (transaction.IsCommitted)
                {
                    throw new InvalidOperationException("Transation is commited.");
                }
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override void Dispose(bool manual, bool wasDisposed)
            {
                if (!wasDisposed && _layer != null && !_layer.Disposable.IsDisposed)
                {
                    if (!IsCommitted && !IsReverted)
                    {
                        Revert();
                    }
                    _layer.ResetCurrentTransation();
                }
            }
        }

        public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);

        private readonly string _logAddress;
        private readonly FileStream _log;
        private readonly Queue<Transition> _transitions;
        private readonly UniqueTimestampFactory _uniqueTimestampFactory;
        private Task _transitionsPusher;
        private Transition _lastCommitedTransition;
        private ulong _currentTransactionId;
        private Queue<Transition> _currentTransactionTransitions;
        private Transaction _currentTransaction;
        private ulong _lastCommitedTransactionId;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
            : base(links)
        {
            if (string.IsNullOrWhiteSpace(logAddress))
            {
                throw new ArgumentNullException(nameof(logAddress));
            }
            // В первой строке файла хранится последняя закоммиченную транзакцию.
            // При запуске это используется для проверки удачного закрытия файла лога.
            // In the first line of the file the last committed transaction is stored.
            // On startup, this is used to check that the log file is successfully closed.
            var lastCommitedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
            var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
            if (!lastCommitedTransition.Equals(lastWrittenTransition))
            {
                Dispose();
```

```
269                 throw new NotSupportedException("Database is damaged, autorecovery is not
                    ↪  supported yet.");
270             }
271             if (lastCommitedTransition == default)
272             {
273                 FileHelpers.WriteFirst(logAddress, lastCommitedTransition);
274             }
275             _lastCommitedTransition = lastCommitedTransition;
276             // TODO: Think about a better way to calculate or store this value
277             var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
278             _lastCommitedTransactionId = allTransitions.Length > 0 ? allTransitions.Max(x =>
                    ↪  x.TransactionId) : 0;
279             _uniqueTimestampFactory = new UniqueTimestampFactory();
280             _logAddress = logAddress;
281             _log = FileHelpers.Append(logAddress);
282             _transitions = new Queue<Transition>();
283             _transitionsPusher = new Task(TransitionsPusher);
284             _transitionsPusher.Start();
285         }
286 
287         [MethodImpl(MethodImplOptions.AggressiveInlining)]
288         public IList<ulong> GetLinkValue(ulong link) => _links.GetLink(link);
289 
290         [MethodImpl(MethodImplOptions.AggressiveInlining)]
291         public override ulong Create(IList<ulong> restrictions)
292         {
293             var createdLinkIndex = _links.Create();
294             var createdLink = new Link<ulong>(_links.GetLink(createdLinkIndex));
295             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                    ↪  default, createdLink));
296             return createdLinkIndex;
297         }
298 
299         [MethodImpl(MethodImplOptions.AggressiveInlining)]
300         public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
301         {
302             var linkIndex = restrictions[_constants.IndexPart];
303             var beforeLink = new Link<ulong>(_links.GetLink(linkIndex));
304             linkIndex = _links.Update(restrictions, substitution);
305             var afterLink = new Link<ulong>(_links.GetLink(linkIndex));
306             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                    ↪  beforeLink, afterLink));
307             return linkIndex;
308         }
309 
310         [MethodImpl(MethodImplOptions.AggressiveInlining)]
311         public override void Delete(IList<ulong> restrictions)
312         {
313             var link = restrictions[_constants.IndexPart];
314             var deletedLink = new Link<ulong>(_links.GetLink(link));
315             _links.Delete(link);
316             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                    ↪  deletedLink, default));
317         }
318 
319         [MethodImpl(MethodImplOptions.AggressiveInlining)]
320         private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
                ↪  _transitions;
321 
322         [MethodImpl(MethodImplOptions.AggressiveInlining)]
323         private void CommitTransition(Transition transition)
324         {
325             if (_currentTransaction != null)
326             {
327                 Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
328             }
329             var transitions = GetCurrentTransitions();
330             transitions.Enqueue(transition);
331         }
332 
333         [MethodImpl(MethodImplOptions.AggressiveInlining)]
334         private void RevertTransition(Transition transition)
335         {
336             if (transition.After.IsNull()) // Revert Deletion with Creation
337             {
338                 _links.Create();
339             }
340             else if (transition.Before.IsNull()) // Revert Creation with Deletion
```

```
            {
                _links.Delete(transition.After.Index);
            }
            else // Revert Update
            {
                _links.Update(new[] { transition.After.Index, transition.Before.Source,
                ↪  transition.Before.Target });
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void ResetCurrentTransation()
        {
            _currentTransactionId = 0;
            _currentTransactionTransitions = null;
            _currentTransaction = null;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void PushTransitions()
        {
            if (_log == null || _transitions == null)
            {
                return;
            }
            for (var i = 0; i < _transitions.Count; i++)
            {
                var transition = _transitions.Dequeue();

                _log.Write(transition);
                _lastCommitedTransition = transition;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void TransitionsPusher()
        {
            while (!Disposable.IsDisposed && _transitionsPusher != null)
            {
                Thread.Sleep(DefaultPushDelay);
                PushTransitions();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Transaction BeginTransaction() => new Transaction(this);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void DisposeTransitions()
        {
            try
            {
                var pusher = _transitionsPusher;
                if (pusher != null)
                {
                    _transitionsPusher = null;
                    pusher.Wait();
                }
                if (_transitions != null)
                {
                    PushTransitions();
                }
                _log.DisposeIfPossible();
                FileHelpers.WriteFirst(_logAddress, _lastCommitedTransition);
            }
            catch (Exception ex)
            {
                ex.Ignore();
            }
        }

        #region DisposalBase

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                DisposeTransitions();
```

```
419             }
420             base.Dispose(manual, wasDisposed);
421         }
422 
423         #endregion
424     }
425 }
```

1.116 ./csharp/Platform.Data.Doublets.Tests/GenericLinksTests.cs

```
1   using System;
2   using Xunit;
3   using Platform.Reflection;
4   using Platform.Memory;
5   using Platform.Scopes;
6   using Platform.Data.Doublets.Memory.United.Generic;
7   
8   namespace Platform.Data.Doublets.Tests
9   {
10      public unsafe static class GenericLinksTests
11      {
12          [Fact]
13          public static void CRUDTest()
14          {
15              Using<byte>(links => links.TestCRUDOperations());
16              Using<ushort>(links => links.TestCRUDOperations());
17              Using<uint>(links => links.TestCRUDOperations());
18              Using<ulong>(links => links.TestCRUDOperations());
19          }
20  
21          [Fact]
22          public static void RawNumbersCRUDTest()
23          {
24              Using<byte>(links => links.TestRawNumbersCRUDOperations());
25              Using<ushort>(links => links.TestRawNumbersCRUDOperations());
26              Using<uint>(links => links.TestRawNumbersCRUDOperations());
27              Using<ulong>(links => links.TestRawNumbersCRUDOperations());
28          }
29  
30          [Fact]
31          public static void MultipleRandomCreationsAndDeletionsTest()
32          {
33              Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
                ↪  MultipleRandomCreationsAndDeletions(16)); // Cannot use more because current
                ↪  implementation of tree cuts out 5 bits from the address space.
34              Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Te
                ↪  stMultipleRandomCreationsAndDeletions(100));
35              Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
                ↪  MultipleRandomCreationsAndDeletions(100));
36              Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
                ↪  tMultipleRandomCreationsAndDeletions(100));
37          }
38  
39          private static void Using<TLink>(Action<ILinks<TLink>> action)
40          {
41              using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪  UnitedMemoryLinks<TLink>>>())
42              {
43                  action(scope.Use<ILinks<TLink>>());
44              }
45          }
46      }
47  }
```

1.117 ./csharp/Platform.Data.Doublets.Tests/LinksConstantsTests.cs

```
1   using Xunit;
2   
3   namespace Platform.Data.Doublets.Tests
4   {
5       public static class LinksConstantsTests
6       {
7           [Fact]
8           public static void ExternalReferencesTest()
9           {
10              LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
                ↪  (long.MaxValue + 1UL, ulong.MaxValue));
11  
12              //var minimum = new Hybrid<ulong>(0, isExternal: true);
13              var minimum = new Hybrid<ulong>(1, isExternal: true);
```

```
            var maximum = new Hybrid<ulong>(long.MaxValue, isExternal: true);

            Assert.True(constants.IsExternalReference(minimum));
            Assert.True(constants.IsExternalReference(maximum));
        }
    }
}
```

1.118 ./csharp/Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs

```
using System.IO;
using Xunit;
using Platform.Singletons;
using Platform.Memory;
using Platform.Data.Doublets.Memory.United.Specific;

namespace Platform.Data.Doublets.Tests
{
    public static class ResizableDirectMemoryLinksTests
    {
        private static readonly LinksConstants<ulong> _constants =
        ↪ Default<LinksConstants<ulong>>.Instance;

        [Fact]
        public static void BasicFileMappedMemoryTest()
        {
            var tempFilename = Path.GetTempFileName();
            using (var memoryAdapter = new UInt64UnitedMemoryLinks(tempFilename))
            {
                memoryAdapter.TestBasicMemoryOperations();
            }
            File.Delete(tempFilename);
        }

        [Fact]
        public static void BasicHeapMemoryTest()
        {
            using (var memory = new
            ↪ HeapResizableDirectMemory(UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
            using (var memoryAdapter = new UInt64UnitedMemoryLinks(memory,
            ↪ UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
            {
                memoryAdapter.TestBasicMemoryOperations();
            }
        }

        private static void TestBasicMemoryOperations(this ILinks<ulong> memoryAdapter)
        {
            var link = memoryAdapter.Create();
            memoryAdapter.Delete(link);
        }

        [Fact]
        public static void NonexistentReferencesHeapMemoryTest()
        {
            using (var memory = new
            ↪ HeapResizableDirectMemory(UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
            using (var memoryAdapter = new UInt64UnitedMemoryLinks(memory,
            ↪ UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
            {
                memoryAdapter.TestNonexistentReferences();
            }
        }

        private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
        {
            var link = memoryAdapter.Create();
            memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
            var resultLink = _constants.Null;
            memoryAdapter.Each(foundLink =>
            {
                resultLink = foundLink[_constants.IndexPart];
                return _constants.Break;
            }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
            Assert.True(resultLink == link);
            Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
            memoryAdapter.Delete(link);
        }
    }
}
```

1.119 ./csharp/Platform.Data.Doublets.Tests/ScopeTests.cs

```
using Xunit;
using Platform.Scopes;
using Platform.Memory;
using Platform.Data.Doublets.Decorators;
using Platform.Reflection;
using Platform.Data.Doublets.Memory.United.Generic;
using Platform.Data.Doublets.Memory.United.Specific;

namespace Platform.Data.Doublets.Tests
{
    public static class ScopeTests
    {
        [Fact]
        public static void SingleDependencyTest()
        {
            using (var scope = new Scope())
            {
                scope.IncludeAssemblyOf<IMemory>();
                var instance = scope.Use<IDirectMemory>();
                Assert.IsType<HeapResizableDirectMemory>(instance);
            }
        }

        [Fact]
        public static void CascadeDependencyTest()
        {
            using (var scope = new Scope())
            {
                scope.Include<TemporaryFileMappedResizableDirectMemory>();
                scope.Include<UInt64UnitedMemoryLinks>();
                var instance = scope.Use<ILinks<ulong>>();
                Assert.IsType<UInt64UnitedMemoryLinks>(instance);
            }
        }

        [Fact(Skip = "Would be fixed later.")]
        public static void FullAutoResolutionTest()
        {
            using (var scope = new Scope(autoInclude: true, autoExplore: true))
            {
                var instance = scope.Use<UInt64Links>();
                Assert.IsType<UInt64Links>(instance);
            }
        }

        [Fact]
        public static void TypeParametersTest()
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪  UnitedMemoryLinks<ulong>>>())
            {
                var links = scope.Use<ILinks<ulong>>();
                Assert.IsType<UnitedMemoryLinks<ulong>>(links);
            }
        }
    }
}
```

1.120 ./csharp/Platform.Data.Doublets.Tests/SplitMemoryGenericLinksTests.cs

```
using System;
using Xunit;
using Platform.Memory;
using Platform.Data.Doublets.Memory.Split.Generic;
using Platform.Data.Doublets.Memory;

namespace Platform.Data.Doublets.Tests
{
    public unsafe static class SplitMemoryGenericLinksTests
    {
        [Fact]
        public static void CRUDTest()
        {
            Using<byte>(links => links.TestCRUDOperations());
            Using<ushort>(links => links.TestCRUDOperations());
            Using<uint>(links => links.TestCRUDOperations());
            Using<ulong>(links => links.TestCRUDOperations());
        }

```

```
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
38          private static void Using<TLink>(Action<ILinks<TLink>> action)
39          {
40              using (var dataMemory = new HeapResizableDirectMemory())
41              using (var indexMemory = new HeapResizableDirectMemory())
42              using (var memory = new SplitMemoryLinks<TLink>(dataMemory, indexMemory))
43              {
44                  action(memory);
45              }
46          }
47
48          private static void UsingWithExternalReferences<TLink>(Action<ILinks<TLink>> action)
49          {
50              var contants = new LinksConstants<TLink>(enableExternalReferencesSupport: true);
51              using (var dataMemory = new HeapResizableDirectMemory())
52              using (var indexMemory = new HeapResizableDirectMemory())
53              using (var memory = new SplitMemoryLinks<TLink>(dataMemory, indexMemory,
                ↪ SplitMemoryLinks<TLink>.DefaultLinksSizeStep, contants))
54              {
55                  action(memory);
56              }
57          }
58      }
59  }
```

1.121 ./csharp/Platform.Data.Doublets.Tests/SplitMemoryUInt32LinksTests.cs

```
1   using System;
2   using Xunit;
3   using Platform.Memory;
4   using Platform.Data.Doublets.Memory.Split.Specific;
5   using TLink = System.UInt32;
6
7   namespace Platform.Data.Doublets.Tests
8   {
9       public unsafe static class SplitMemoryUInt32LinksTests
10      {
11          [Fact]
12          public static void CRUDTest()
13          {
14              Using(links => links.TestCRUDOperations());
15          }
16
17          [Fact]
18          public static void RawNumbersCRUDTest()
19          {
20              UsingWithExternalReferences(links => links.TestRawNumbersCRUDOperations());
21          }
22
23          [Fact]
24          public static void MultipleRandomCreationsAndDeletionsTest()
25          {
26              Using(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultip
                ↪ leRandomCreationsAndDeletions(500));
27          }
28
29          private static void Using(Action<ILinks<TLink>> action)
```

```
        {
            using (var dataMemory = new HeapResizableDirectMemory())
            using (var indexMemory = new HeapResizableDirectMemory())
            using (var memory = new UInt32SplitMemoryLinks(dataMemory, indexMemory))
            {
                action(memory);
            }
        }

        private static void UsingWithExternalReferences(Action<ILinks<TLink>> action)
        {
            var contants = new LinksConstants<TLink>(enableExternalReferencesSupport: true);
            using (var dataMemory = new HeapResizableDirectMemory())
            using (var indexMemory = new HeapResizableDirectMemory())
            using (var memory = new UInt32SplitMemoryLinks(dataMemory, indexMemory,
            ↪  UInt32SplitMemoryLinks.DefaultLinksSizeStep, contants))
            {
                action(memory);
            }
        }
    }
}
```

1.122 ./csharp/Platform.Data.Doublets.Tests/SplitMemoryUInt64LinksTests.cs

```
using System;
using Xunit;
using Platform.Memory;
using Platform.Data.Doublets.Memory.Split.Specific;
using TLink = System.UInt64;

namespace Platform.Data.Doublets.Tests
{
    public unsafe static class SplitMemoryUInt64LinksTests
    {
        [Fact]
        public static void CRUDTest()
        {
            Using(links => links.TestCRUDOperations());
        }

        [Fact]
        public static void RawNumbersCRUDTest()
        {
            UsingWithExternalReferences(links => links.TestRawNumbersCRUDOperations());
        }

        [Fact]
        public static void MultipleRandomCreationsAndDeletionsTest()
        {
            Using(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultip
            ↪  leRandomCreationsAndDeletions(500));
        }

        private static void Using(Action<ILinks<TLink>> action)
        {
            using (var dataMemory = new HeapResizableDirectMemory())
            using (var indexMemory = new HeapResizableDirectMemory())
            using (var memory = new UInt64SplitMemoryLinks(dataMemory, indexMemory))
            {
                action(memory);
            }
        }

        private static void UsingWithExternalReferences(Action<ILinks<TLink>> action)
        {
            var contants = new LinksConstants<TLink>(enableExternalReferencesSupport: true);
            using (var dataMemory = new HeapResizableDirectMemory())
            using (var indexMemory = new HeapResizableDirectMemory())
            using (var memory = new UInt64SplitMemoryLinks(dataMemory, indexMemory,
            ↪  UInt64SplitMemoryLinks.DefaultLinksSizeStep, contants))
            {
                action(memory);
            }
        }
    }
}
```

1.123 ./csharp/Platform.Data.Doublets.Tests/TestExtensions.cs

```
using System.Collections.Generic;
using Xunit;
using Platform.Ranges;
using Platform.Numbers;
using Platform.Random;
using Platform.Setters;
using Platform.Converters;

namespace Platform.Data.Doublets.Tests
{
    public static class TestExtensions
    {
        public static void TestCRUDOperations<T>(this ILinks<T> links)
        {
            var constants = links.Constants;

            var equalityComparer = EqualityComparer<T>.Default;

            var zero = default(T);
            var one = Arithmetic.Increment(zero);

            // Create Link
            Assert.True(equalityComparer.Equals(links.Count(), zero));

            var setter = new Setter<T>(constants.Null);
            links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);

            Assert.True(equalityComparer.Equals(setter.Result, constants.Null));

            var linkAddress = links.Create();

            var link = new Link<T>(links.GetLink(linkAddress));

            Assert.True(link.Count == 3);
            Assert.True(equalityComparer.Equals(link.Index, linkAddress));
            Assert.True(equalityComparer.Equals(link.Source, constants.Null));
            Assert.True(equalityComparer.Equals(link.Target, constants.Null));

            Assert.True(equalityComparer.Equals(links.Count(), one));

            // Get first link
            setter = new Setter<T>(constants.Null);
            links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);

            Assert.True(equalityComparer.Equals(setter.Result, linkAddress));

            // Update link to reference itself
            links.Update(linkAddress, linkAddress, linkAddress);

            link = new Link<T>(links.GetLink(linkAddress));

            Assert.True(equalityComparer.Equals(link.Source, linkAddress));
            Assert.True(equalityComparer.Equals(link.Target, linkAddress));

            // Update link to reference null (prepare for delete)
            var updated = links.Update(linkAddress, constants.Null, constants.Null);

            Assert.True(equalityComparer.Equals(updated, linkAddress));

            link = new Link<T>(links.GetLink(linkAddress));

            Assert.True(equalityComparer.Equals(link.Source, constants.Null));
            Assert.True(equalityComparer.Equals(link.Target, constants.Null));

            // Delete link
            links.Delete(linkAddress);

            Assert.True(equalityComparer.Equals(links.Count(), zero));

            setter = new Setter<T>(constants.Null);
            links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);

            Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
        }

        public static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
        {
            // Constants
            var constants = links.Constants;
            var equalityComparer = EqualityComparer<T>.Default;
```

```
81
82              var zero = default(T);
83              var one = Arithmetic.Increment(zero);
84              var two = Arithmetic.Increment(one);
85
86              var h106E = new Hybrid<T>(106L, isExternal: true);
87              var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
88              var h108E = new Hybrid<T>(-108L);
89
90              Assert.Equal(106L, h106E.AbsoluteValue);
91              Assert.Equal(107L, h107E.AbsoluteValue);
92              Assert.Equal(108L, h108E.AbsoluteValue);
93
94              // Create Link (External -> External)
95              var linkAddress1 = links.Create();
96
97              links.Update(linkAddress1, h106E, h108E);
98
99              var link1 = new Link<T>(links.GetLink(linkAddress1));
100
101             Assert.True(equalityComparer.Equals(link1.Source, h106E));
102             Assert.True(equalityComparer.Equals(link1.Target, h108E));
103
104             // Create Link (Internal -> External)
105             var linkAddress2 = links.Create();
106
107             links.Update(linkAddress2, linkAddress1, h108E);
108
109             var link2 = new Link<T>(links.GetLink(linkAddress2));
110
111             Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
112             Assert.True(equalityComparer.Equals(link2.Target, h108E));
113
114             // Create Link (Internal -> Internal)
115             var linkAddress3 = links.Create();
116
117             links.Update(linkAddress3, linkAddress1, linkAddress2);
118
119             var link3 = new Link<T>(links.GetLink(linkAddress3));
120
121             Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
122             Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));
123
124             // Search for created link
125             var setter1 = new Setter<T>(constants.Null);
126             links.Each(h106E, h108E, setter1.SetAndReturnFalse);
127
128             Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));
129
130             // Search for nonexistent link
131             var setter2 = new Setter<T>(constants.Null);
132             links.Each(h106E, h107E, setter2.SetAndReturnFalse);
133
134             Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));
135
136             // Update link to reference null (prepare for delete)
137             var updated = links.Update(linkAddress3, constants.Null, constants.Null);
138
139             Assert.True(equalityComparer.Equals(updated, linkAddress3));
140
141             link3 = new Link<T>(links.GetLink(linkAddress3));
142
143             Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
144             Assert.True(equalityComparer.Equals(link3.Target, constants.Null));
145
146             // Delete link
147             links.Delete(linkAddress3);
148
149             Assert.True(equalityComparer.Equals(links.Count(), two));
150
151             var setter3 = new Setter<T>(constants.Null);
152             links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);
153
154             Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
155         }
156
157         public static void TestMultipleRandomCreationsAndDeletions<TLink>(this ILinks<TLink>
            ↪ links, int maximumOperationsPerCycle)
158         {
159             var comparer = Comparer<TLink>.Default;
```

```
            var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
            var uInt64ToAddressConverter = CheckedConverter<ulong, TLink>.Default;
            for (var N = 1; N < maximumOperationsPerCycle; N++)
            {
                var random = new System.Random(N);
                var created = 0UL;
                var deleted = 0UL;
                for (var i = 0; i < N; i++)
                {
                    var linksCount = addressToUInt64Converter.Convert(links.Count());
                    var createPoint = random.NextBoolean();
                    if (linksCount >= 2 && createPoint)
                    {
                        var linksAddressRange = new Range<ulong>(1, linksCount);
                        TLink source = uInt64ToAddressConverter.Convert(random.NextUInt64(linksA
                        ↪  ddressRange));
                        TLink target = uInt64ToAddressConverter.Convert(random.NextUInt64(linksA
                        ↪  ddressRange));
                        ↪  //-V3086
                        var resultLink = links.GetOrCreate(source, target);
                        if (comparer.Compare(resultLink,
                        ↪  uInt64ToAddressConverter.Convert(linksCount)) > 0)
                        {
                            created++;
                        }
                    }
                    else
                    {
                        links.Create();
                        created++;
                    }
                }
                Assert.True(created == addressToUInt64Converter.Convert(links.Count()));
                for (var i = 0; i < N; i++)
                {
                    TLink link = uInt64ToAddressConverter.Convert((ulong)i + 1UL);
                    if (links.Exists(link))
                    {
                        links.Delete(link);
                        deleted++;
                    }
                }
                Assert.True(addressToUInt64Converter.Convert(links.Count()) == 0L);
            }
        }
    }
}
```

1.124 ./csharp/Platform.Data.Doublets.Tests/Uint64LinksExtensionsTests.cs

```
using Platform.Data.Doublets.Memory;
using Platform.Data.Doublets.Memory.United.Generic;
using Platform.Data.Numbers.Raw;
using Platform.Memory;
using Platform.Numbers;
using Xunit;
using Xunit.Abstractions;
using TLink = System.UInt64;

namespace Platform.Data.Doublets.Tests
{
    public class Uint64LinksExtensionsTests
    {
        public static ILinks<TLink> CreateLinks() => CreateLinks<TLink>(new
        ↪  Platform.IO.TemporaryFile());

        public static ILinks<TLink> CreateLinks<TLink>(string dataDBFilename)
        {
            var linksConstants = new LinksConstants<TLink>(enableExternalReferencesSupport:
            ↪  true);
            return new UnitedMemoryLinks<TLink>(new
            ↪  FileMappedResizableDirectMemory(dataDBFilename),
            ↪  UnitedMemoryLinks<TLink>.DefaultLinksSizeStep, linksConstants,
            ↪  IndexTreeType.Default);
        }
        [Fact]
        public void FormatStructureWithExternalReferenceTest()
        {
            ILinks<TLink> links = CreateLinks();
            TLink zero = default;
```

```
            var one = Arithmetic.Increment(zero);
            var markerIndex = one;
            var meaningRoot = links.GetOrCreate(markerIndex, markerIndex);
            var numberMarker = links.GetOrCreate(meaningRoot, Arithmetic.Increment(ref
            ↪  markerIndex));
            AddressToRawNumberConverter<TLink> addressToNumberConverter = new();
            var numberAddress = addressToNumberConverter.Convert(1);
            var numberLink = links.GetOrCreate(numberMarker, numberAddress);
            var linkNotation = links.FormatStructure(numberLink, link => link.IsFullPoint(),
            ↪  true);
            Assert.Equal("(3:(2:1 2) 18446744073709551615)", linkNotation);
        }
    }
}
```

1.125 ./csharp/Platform.Data.Doublets.Tests/UnitedMemoryUInt32LinksTests.cs

```
using System;
using Xunit;
using Platform.Reflection;
using Platform.Memory;
using Platform.Scopes;
using Platform.Data.Doublets.Memory.United.Specific;
using TLink = System.UInt32;

namespace Platform.Data.Doublets.Tests
{
    public unsafe static class UnitedMemoryUInt32LinksTests
    {
        [Fact]
        public static void CRUDTest()
        {
            Using(links => links.TestCRUDOperations());
        }

        [Fact]
        public static void RawNumbersCRUDTest()
        {
            Using(links => links.TestRawNumbersCRUDOperations());
        }

        [Fact]
        public static void MultipleRandomCreationsAndDeletionsTest()
        {
            Using(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultip
            ↪  leRandomCreationsAndDeletions(100));
        }

        private static void Using(Action<ILinks<TLink>> action)
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪  UInt32UnitedMemoryLinks>>())
            {
                action(scope.Use<ILinks<TLink>>());
            }
        }
    }
}
```

1.126 ./csharp/Platform.Data.Doublets.Tests/UnitedMemoryUInt64LinksTests.cs

```
using System;
using Xunit;
using Platform.Reflection;
using Platform.Memory;
using Platform.Scopes;
using Platform.Data.Doublets.Memory.United.Specific;
using TLink = System.UInt64;

namespace Platform.Data.Doublets.Tests
{
    public unsafe static class UnitedMemoryUInt64LinksTests
    {
        [Fact]
        public static void CRUDTest()
        {
            Using(links => links.TestCRUDOperations());
        }

        [Fact]
        public static void RawNumbersCRUDTest()
```

```
        {
            Using(links => links.TestRawNumbersCRUDOperations());
        }

        [Fact]
        public static void MultipleRandomCreationsAndDeletionsTest()
        {
            Using(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultip
            ↪ leRandomCreationsAndDeletions(100));
        }

        private static void Using(Action<ILinks<TLink>> action)
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪ UInt64UnitedMemoryLinks>>())
            {
                action(scope.Use<ILinks<TLink>>());
            }
        }
    }
}
```

Index